CROSSTALK

July / August 2014 The Journal of Defense Software Engineering Vol. 27 No. 4

HIGH MATURITY ORGANIZATIONAL CHARACTERISTICS

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
JUL 2014		**00-00-2014 to 00-00-2014**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CrossTalk, The Journal of Defense Software Engineering. Volume 27, Number 4. July/August 2014	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **517 SMXS/MXDED,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**44**	

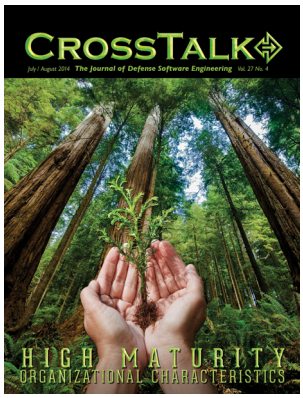
TABLE OF CONTENTS

Departments

3 From the Sponsor

41 Upcoming Events

43 BackTalk



Cover Design by Kent Bingham

High Maturity Organizational Characteristics

4 **Agile and the Definition of Quality**
Newcomers to Agile often fail to see the connection between older thoughts about Agile and today's Agile movement.
by Gerald M. Weinberg

8 **High Maturity Is Not A Procrustean Bed**
Too many organizations have a single model of high maturity to which they try to fit all their projects.
by Barry Boehm, Richard Turner, Jo Ann Lane, and Supannika Koolmanojwong

15 **Disciplined Learning: The Successor to Risk Management**
Disciplined learning, or "learn early, learn often," updates naïve agile development and traditional risk management, and safely replaces the dreaded catch phrase, "fail early fail often."
by Alistair Cockburn

19 **Achieving Software Excellence**
Software is the main operational component of every major organization in the world, but software quality is still not acceptable for many applications.
by Capers Jones

26 **Improving Software through Metrics while Providing Cradle to Grave Support Metrics** are beneficial to an organization that supports a product from inception through product retirement and disposal.
by Jennifer Walters, and Kevin MacG. Adams, Ph.D.

30 **Paths of Adoption: Routes to Continuous Process Improvement**
The long-term goals of Process Improvement should be to introduce and sustain a culture of continuous process improvement.
by David Saint-Amand and Mark Stockmyer

36 **High Maturity Heresy**
How rocket scientists implement High Maturity.
by Tom Lienhard

CROSSTALK

NAVAIR Jeff Schwalb
DHS Joe Jarzombek
309 SMXG Karl Rogers

Publisher Justin T. Hill
Article Coordinator Heather Giacalone
Managing Director David Erickson
Technical Program Lead Thayne M. Hill
Managing Editor Brandon Ellis
Associate Editor Colin Kelly
Art Director Kevin Kiernan

Phone 801-777-9828
E-mail Crosstalk.Articles@hill.af.mil
CrossTalk Online www.crosstalkonline.org

CROSSTALK, The Journal of Defense Software Engineering is co-sponsored by the U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). USN co-sponsor: Naval Air Systems Command. USAF co-sponsor: Ogden-ALC 309 SMXG. DHS co-sponsor: Office of Cybersecurity and Communications in the National Protection and Programs Directorate.

The USAF Software Technology Support Center (STSC) is the publisher of CROSSTALK providing both editorial oversight and technical review of the journal. CROSSTALK'S mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

Subscriptions: Visit <www.crosstalkonline.org/subscribe> to receive an e-mail notification when each new issue is published online or to subscribe to an RSS notification feed.

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CROSSTALK editorial board prior to publication. Please follow the Author Guidelines, available at <www.crosstalkonline.org/submission-guidelines>. CROSSTALK does not pay for submissions. Published articles remain the property of the authors and may be submitted to other publications. Security agency releases, clearances, and public affairs office approvals are the sole responsibility of the authors and their organizations.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CROSSTALK.

Trademarks and Endorsements: CROSSTALK is an authorized publication for members of the DoD. Contents of CROSSTALK are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CROSSTALK Online Services:
For questions or concerns about crosstalkonline.org web content or functionality contact the CROSSTALK webmaster at 801-417-3000 or webmaster@luminpublishing.com.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

CROSSTALK is published six times a year by the U.S. Air Force STSC in concert with Lumin Publishing <luminpublishing.com>.
ISSN 2160-1577 (print); ISSN 2160-1593 (online)

CrossTalk would like to thank 309 SMXG for sponsoring this issue.

Anyone who has spent a significant amount of time working toward organizational and process high maturity knows that it is never an easy or short-term endeavor. This issue of CrossTalk focuses on High Maturity Organizational Characteristics. The complexities of creating and sustaining a High Maturity Organization never cease to amaze me. There are many ways to achieve High Maturity and this issue focuses on many of the perspectives from people who have worked in the software industry for many years.

Gerald Weinberg sometimes called the Father of Agile, focuses on Agile and the often-elusive definition of Quality. When it comes to software, what is Quality and how good is good enough?

Barry Boehm, Richard Turner, Jo Ann Lane, and Supannika Koolmanojwong focus on the fact that high maturity processes are not one size fits all in their article, "High Maturity is Not a Procrustean Bed." Due to the complexity of high maturity processes and tools it is tempting for many organizations to try and make their process a one size fits all for high maturity projects, this approach can be problematic however there are ways to determine which process, or processes best fit a particular project.

In his article, "Disciplined Learning: The Successor to Risk Management" Alistair Cockburn, investigates the idea that traditional Risk Management is focused on avoiding failure and not delivering success. He investigates how Disciplined Learning may add to the probability of success of programs.

Capers Jones provides insight into proven methods of achieving excellence in software development in his article, "Achieving Software Excellence." He also explores the definition of what software excellence really means.

"Improving Software through Metrics while Providing Cradle to Grave Support" was written by Jennifer Walters and Kevin MacG. Adams. This article focuses on the benefits of metrics collection over the lifecycle of the system. Above are examples of the articles in this issue of CrossTalk Magazine. As I write this article I just walked out of a CMMI® High Maturity SCAMPI B appraisal. It was very interesting and informative. It reminded me once again that High Maturity is an involved and difficult process. I have consistently found that the more we learn about processes and all of the aspects of software process improvement the more able we are to lead our organizations/projects to provide high quality software on schedule, at a reasonable cost. I hope you enjoy this edition of CrossTalk Magazine and always continue to learn.

Disclaimer:

CMMI® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Karl G. Rogers
Software Maintenance Group Director
309th Software Maintenance Group

Agile and the Definition of Quality

Gerald M. Weinberg, Author

Abstract. To convince people of the value of Agile, we need to produce new software that is full of wonderful features that the old software didn't possess while functioning exactly the way as the old software did.

Introduction

Some Agile writers have called me "the grandfather of Agile." I choose to interpret that comment as a compliment, rather than a disparagement of my advanced age. As a grandfather, much of my most influential writing was done long before the Agile movement appeared on stage. As a result, newcomers on the scene often fail to see the connection between those writings and today's Agile movement.

I use my blog to correct that situation, with a series of articles relating specific material to Agile basics. I started with an essay about my definition of "quality"—often quoted by not always understood.

A Bug in the Family

My sister's daughter, Terra, is the only one in the family who has followed Uncle Jerry in the writer's trade. She writes fascinating books on the history of medicine, and I follow each one's progress as if it were one of my own. For that reason, I was terribly distressed when her first book, Disease in the Popular American Press, came out with a number of gross typographical errors in which whole segments of text disappeared. I was even more distressed to discover that those errors were caused by an error in the word processing software she used—CozyWrite, published by one of my clients, the MiniCozy Software Company.

Terra asked me to discuss the matter with MiniCozy on my next visit. I located the project manager for CozyWrite, and he acknowledged the existence of the error.

"It's a rare bug," he said.

"I wouldn't say so," I countered. "I found over twenty-five instances in her book."

"But it would only happen in a book-sized project. Out of over 100,000 customers, we probably didn't have 10 who undertook a project of that size as a single file."

"But my niece noticed. It was her first book, and she was devastated."

"Naturally I'm sorry for her, but it wouldn't have made any sense for us to try to fix the bug for 10 customers."

"Why not? You advertise that CozyWrite handles book-sized projects."

"We tried to do that, but the features didn't work. Eventually, we'll probably fix them, but for now, chances are we would introduce a worse bug—one that would affect hundreds or thousands of customers. I believe we did the right thing."

As I listened to this project manager, I found myself caught in an emotional trap. As software consultant to MiniCozy, I had to agree, but as uncle to an author, I was violently opposed to his line of reasoning. If someone at that moment had asked me, "Is CozyWrite a quality product?" I would have been tongue-tied.

How would you have answered?

The Relativity of Quality

The reason for my dilemma lies in the relativity of quality. As the MiniCozy story crisply illustrates, what is adequate quality to one person may be inadequate quality to another.

Finding the Relativity

If you examine various definitions of quality, you will always find this relativity. You may have to examine with care, though, for the relativity is often hidden, or at best, implicit.

Take for example Crosby's definition:

"Quality is meeting requirements."

Unless your requirements come directly from heaven (as some developers seem to think), a more precise statement would be:

"Quality is meeting some person's requirements."

For each different person, the same product will generally have different "quality," as in the case of my niece's word processor. My MiniCozy dilemma is resolved once I recognize two things:

a. To Terra, the people involved were her readers.

b. To MiniCozy's project manager, the people involved were (the majority of) his customers.

Who Was That Masked Man?

In short, quality does not exist in a non-human vacuum, but every statement about quality is a statement about some person(s).

That statement about quality may be explicit or implicit. Most often, the "who" is implicit, and statements about quality sound like something Moses brought down from Mount Sinai on a stone tablet. That's why so many discussions of software quality are unproductive—it's my stone tablet versus your Golden Calf.

When we encompass the relativity of quality, we have a tool to make those discussions more fruitful. Each time somebody asserts a definition of software quality, we simply ask, "Who is the person behind that statement about quality."

Using this heuristic, let's consider a few familiar but often conflicting ideas about what constitutes software quality:

a. "Zero defects is high quality"

1. to a user such as a surgeon whose work would be disturbed by those defects
2. to a manager who would be criticized for those defects

b. "Lots of features is high quality"

1. to users whose work can use those features–if they know about them
2. to marketers who believe that features sell products

c. "Elegant coding is high quality"
1. to developers who place a high value on the opinions of their peers
2. to professors of computer science who enjoy elegance

d. "High performance is high quality"
1. to users whose work taxes the capacity of their machines
2. to salespeople who have to submit their products to benchmarks

e. "Low development cost is high quality"
1. to customers who wish to buy thousands of copies of the software
2. to project managers who are on tight budgets

f. "Rapid development is high quality"
1. to users whose work is waiting for the software
2. to marketers who want to colonize a market before the competitors can get in

g. "User-friendliness is high quality"
1. to users who spend 8 hours a day sitting in front of a screen using the software
2. to users who can't remember interface details from one use to the next

The Political Dilemma

Recognizing the relativity of quality often resolves the semantic dilemma. This is a monumental contribution, but it still does not resolve the political dilemma; More quality for one person may mean less quality for another.

For instance, if our goal were "total quality," we'd have to do a summation over all relevant people. Thus, this "total quality" effort would have to start with a comprehensive requirements process that identifies and involves all relevant people. Then, for each design, for each software engineering approach, we would have to assign a quality measure for each person. Summing these measures would then yield the total quality for each different approach.

In practice, of course, no software development project ever uses such an elaborate process. Instead, most people are eliminated by a prior process that decides whose opinion of quality is to count when making decisions?

For instance, the project manager at MiniCozy decided, without hearing arguments from Terra, that her opinion carried minuscule weight in his "software engineering" decision. From this case, we see that software engineering is not a democratic business. Nor, unfortunately, is it a rational business, for these decisions about "who counts" are generally made on an emotional basis.

Quality Is Value To Some Person

The political/emotional dimension of quality is made evident by a somewhat different definition of quality. The idea of "requirements" is a bit too innocent to be useful in this early stage, because it says nothing about whose requirements count the most. A more workable definition would be this, "Quality is value to some person." By "value," I mean, "What are people willing to pay (or do) to have their requirements met."

Suppose, for instance, that Terra were not my niece, but the niece of the president of the MiniCozy Software Company. Knowing MiniCozy's president's reputation for impulsive emotional action, the project manager might have defined "quality" of the word processor differently. In that case, Terra's opinion would have been given high weight in the decision about which faults to repair.

The Impact on Agile Practices

In short, the definition of "quality" is always political and emotional, because it always involves a series of decisions about whose opinions count, and how much they count relative to one another. Of course, much of the time these political/emotional decisions—like all important political/emotional decisions—are hidden from public view. Most of us software people like to appear rational. That's why very few people appreciate the impact of this definition of quality on the Agile approaches.

What makes our task even more difficult is that most of the time these decisions are hidden even from the conscious minds of the persons who make them. That's why one of the most important actions of an Agile team is bringing such decisions into consciousness, if not always into public awareness. And that's why development teams working with an open process (like Agile) are more likely to arrive at a more sensible definition of quality than one developer working alone. To me, any team with even one secret component is not really Agile.

Customer support is another emphasis in Agile processes, and this definition of quality guides the selection of the "customers." To put it succinctly, the "customer" must actively represent all of the significant definitions of "quality." Any missing component of quality may very likely lead to a product that's deficient in that aspect of quality.

As a consultant to supposedly Agile teams, I always examine whether or not they have active participation of a suitable representation of diverse views of their product's quality. If they tell me, "We can be more agile if we don't have to bother satisfying so many people," then they may indeed be agile, but they're definitely not Agile (capital A).

Why People Don't Instantly Buy Into Agile Methods: A Catch-22

When "selling" their methods, Agile evangelists often stress the strength of Agile methods at removing, and even preventing, errors. I used to do this myself, but I always wondered how people could resist this sales pitch. I would plead, "Don't you want quality?" And, although they always said, "Yes, we want quality," they didn't buy what I was selling. Eventually, I learned the reason, or at least one of the reasons. Why can Agile methods be so difficult to sell?

Another Story About Quality

I've demonstrated how "quality" is relative to particular persons. To test our understanding of this definition, as well as its applicability, let's read another story, one that illustrates that quality is not merely the absence of error.

One of the favorite pastimes of my youth was playing cribbage with my father. Cribbage is a card game, invented by the poet Sir John Suckling, very popular in some regions of the

world, but essentially unknown in others. After my father died, I missed playing cribbage with him and was hard pressed to find a regular partner. Consequently, I was delighted to discover a shareware cribbage program for the Macintosh: "Precision Cribbage" by Doug Brent, of San Jose, CA.

Precision Cribbage was a rather nicely engineered piece of software, I thought, especially when compared with the great majority of shareware. I was especially pleased to find that it gave me a challenging game, though it wasn't good enough to beat me more than 1 or 2 games out of 10.

Doug had requested a postcard from my hometown as a shareware fee. I played many happy games of Precision Cribbage, so I was pleased to send Doug this minimum fee. Soon after I sent the card, though, I discovered two clear errors in the scoring algorithm of Precision Cribbage.

(Perhaps the word "precision" in the name should have been a clue. If it was indeed precise, there was no need to call it "precision." The software would have spoken for itself. I often use that observation about product names to begin my evaluation of a project. For instance, whenever a product has the word "magic" in its title, I steer clear of the whole mess.)

One error in Precision Cribbage was an intermittent failure to count correctly hands with three cards of one denomination and two of another (a "full house," in poker terminology). This was clearly an unintentional flaw, because sometimes such hands were counted correctly.

The second error, however, may have been a misunderstanding of the scoring rules (which were certainly part of the "requirements" for a program that purported to play a card game). It had to do with counting hands that had three cards of the same suit when a fourth card of that suit turned up when the deck was cut. In this case, I could actually prove mathematically that the algorithm was incorrect.

So what makes this story relevant? Simply this: even with two scoring errors in the game, I was sufficiently satisfied with the quality of Precision Cribbage to:

a. keep on playing it, for at least several of my valuable hours each week

b. pay the shareware "fee," even though I could have omitted payment with no fear of retribution of any kind

In short, Precision Cribbage had great value to me, value that I was willing and able to demonstrate by spending my own time and (if requested) money. Moreover, had Doug corrected these errors, it would have added very little to the value of the software.

What's Happening to Quality?

My experience with Precision Cribbage took place some years ago, and occurred in a more-or-less amateur piece of shareware. Certainly, with all we've learned over the past few decades, the rate of software errors has diminished. Or has it?

I've conducted a small survey of more modern software. Software written by professionals. Software I use regularly. Software I paid real money for. And not software for playing games, but software used for serious tasks in my business. Here's what I found:

Out of the 20 apps I use most frequently, 16 have bugs that I have personally encountered—bugs that have cost me at least inconvenience and sometime many hours of fix-up time, but at least one hour for each occurrence. If I value my time at a conservative $100/hour (I actually bill at $500/hour), these bugs cost me approximately $5,000 in the month of August. If I maintain that average, that's $60,000 a year.

If I consider only the purchase prices, those 20 apps cost me about $3,500. In other words, over one year, the purchase price of the software represents less than 10% of what it costs me. (And these are selected apps. The ones that are even buggier have been discarded any time I can find a plausible substitute.) In other words, since quality is value, there's a large negative quality associated with this set of applications.

And that's only for one person. In the USA, there must be at least 100,000,000 users of personal computers. My hourly rate is probably higher than the average, so let's just estimate $10/hour, roughly minimum wage for the average person. That would give us an estimate $6,000/year per person for buggy software, which adds up to about $600,000,000,000 for the annual cost to United States workers. Even if my estimates are way off, that's not chump change.

Why Is Improving Quality So Difficult?

If they payoff is so huge, why aren't we raising software quality to new levels? We could ask the same question about improving auto safety, where tens of thousands of human lives are destroyed every year in the United States. You might think that's more motivation than any number of dollars, but it doesn't work that way. Unless the person killed in the car is someone we know, we've heard about so many traffic deaths that we've grown immune to the terrible cost. In other words, it's precisely because traffic deaths are so common that we don't get awfully excited about them.

And, I believe, it's the same with software failures. They're so common that we've learned to take them with an accepting shrug. We simply reboot and get back to work. Very seldom do we even bother to switch to a different app. The old one, with all its bugs, is too familiar, too comfortable. In fact, some people obtain most of their job security precisely because of their familiarity with software bugs and ways to work around them.

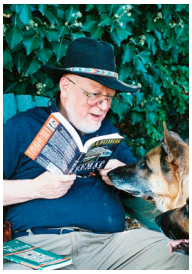
In other words, we're surprised that people don't generally feel motivated to improve quality because we vastly underrate the value of the familiar. And that observation explains an interesting paradox. Agile advocates are often so eager to prove the value of Agile methods that they strive to create products with all sorts of wonderful new features. But each new feature, no matter how potentially valuable, has a downside—a negative quality value because of its unfamiliarity. The harder we strive to produce "higher quality," the lower the quality we tend to produce.

It's a classic catch-22. To convince people of the value of Agile, we need to produce software that is full of wonderful features that the old software didn't possess, at the same time the new software functions exactly the way the old software did. No wonder it's so difficult to change the way we develop software.

Disclaimers:

Author's note: For this article, I've adapted some material from my book, "Agile Impressions" <https://leanpub.com/jerrysblog>

ABOUT THE AUTHOR



Gerald M. Weinberg is the winner of many awards for his writing. His works include "The Psychology of Computer Programming, An Introduction to General Systems Thinking, Becoming a Technical Leader," "Quality Software Series," "Perfect Software," and other books on computing, consulting, human behavior, writing, and technofiction. In addition to his works on software development, Gerald is also science fiction author. He was an architect of NASA's space tracking network and designed and built the first real-time multiprogrammed OS. Gerald is in the University of Nebraska Hall of Fame and is a Founding member of the Computing Hall of Fame.

E-mail: jerryweinberg@comcast.net

High Maturity Is Not A Procrustean Bed

Barry Boehm, Stevens Institute of Technology
Richard Turner, Stevens Institute of Technology
Jo Ann Lane, University of Southern California
Supannika Koolmanojwong, University of Southern California

Abstract. In Greek mythology, Procrustes was a rogue smith and bandit who invited travellers to rest in his "perfectly sized bed." When they accepted, he forcibly bound them to it, then stretched them or cut off various body parts until they "perfectly" fit the bed. Too many organizations have a single model of high maturity to which they try to fit all their projects. Development and acquisition organizations are finding that competitive success requires systems that are a mix of high security assurance components, opaque and dynamic COTS products and cloud services, and highly useful but kaleidoscopic apps and widgets. Approaching such systems with a one-size-fits-all corporate process and maturity model often results in a procrustean fit.

As a process model generator, the Incremental Commitment Spiral Model has a set of criteria for determining which process or processes best fit a particular system of interest. This article summarizes the criteria and illustrates how they have been successfully applied in various situations [1].

Introduction

Too often, high maturity is seen as a proven, standard process that is tailored down or up or in other ways twisted and tortured to adapt to projects that simply don't fit the process. This flies in the face of the definition of a high maturity organization as agile, flexible, and continuously improving. Rapid change, requirements uncertainty, and short capability delivery cycles are increasing the need for such agility, and the traditional process and lifecycle models are not meeting the challenge.

Table 1 describes some examples of Procrustean situations that result from inflexible or overly constrained "high maturity" or otherwise "disciplined" approaches. It elaborates the situation into the likely undesired project result, an example, and a remedy or means of avoiding the situation using the ICSM's four primary principles: Stakeholder value-based guidance; Incremental commitment and accountability; Concurrent multi-discipline engineering; and Evidence and risk-based decisions.

A Different Approach

The Incremental Commitment Spiral Model (ICSM),[1,2] shown in Figure 1, is the result of our efforts to better integrate the hardware, software, and human factors aspects of systems, to provide value to the users as quickly as possible, and to handle the increasingly rapid pace of change. While its pedigree lies in the spiral concept first broadly published in 1988,[3] this new version draws on over 20 years of experience helping people deal with the fact that the original version was too easy to misinterpret.

Fundamental Principles

In hindsight, most of the problems in using the 1988 spiral model came from users constructing processes that had nothing to do with the underlying concepts. The ICSM's four underlying principles, based on observed failure modes over years of experience, are:

Stakeholder value-based guidance. Failing to include and address the value propositions of its success-critical stakeholders can result in their minimal commitment to the project; they may underperform, decline to use, or block the use of the results.

Incremental commitment and accountability. If success-critical stakeholders are not accountable for their commitments (or lack thereof), and the associated consequences (good or bad), they may not provide necessary commitments or decisions in a timely manner and are likely to be drawn away to other pursuits when they are most needed.

Concurrent multi-discipline engineering. Sequential definition and development of a) requirements and solutions; b) hardware, software, and human factors; or c) product and processes likely slows the project and leads to early, hard-to-undo commitments that limit options for project success.

Evidence and risk-based decisions. If key decisions are made based on assertions, vendor literature, or meeting an arbitrary schedule without access to evidence of feasibility, the project is building up risks.

The annual series of "Top-5 Quality Software Projects" software-intensive systems projects published in CrossTalk[4] are examples of successful projects that applied the ICSM principles. These were chosen annually between 2002 and 2005 by panels of leading experts as role models of best practices and successful outcomes. Of the 20 Top-5 projects, 16 explicitly used concurrent engineering; 14 explicitly used risk-driven development; and 15 explicitly used incrementally committed, iterative system evolution. Additional projects gave indications of their partial use. Unfortunately, the project summaries did not include discussion of stakeholder involvement.

The ICSM is not a single one-size-fits-all process. It is actually a process generator, which steers your process in different directions, depending on your particular circumstances. Unlike in the traditional sequential approaches, each spiral concurrently addresses all of the activities of product development to include:

- Requirements (objectives and constraints)
- Solutions (alternatives)
- Products and processes
- Hardware
- Software
- Human factors aspects
- Business case analysis of alternative product configurations
- Product line investments

In this way, ICSM helps adapt your lifecycle strategies and processes to your sources of change. It also supports more rapid system development and evolution through concurrent engineering, enabling you to develop and evolve systems more rapidly and to avoid obsolescence. It is, in many ways, the antithesis of Procrustes bed – one that adjusts to the person, not the other way around.

Table 1. Examples of Procrustean Process Consequences

Issue	Result	Example
Defined process mismatch	Large systems get their integration lopped off in trying to keep to 4-week increments	In their paper "Recognizing and Responding to 'Bad Smells' in Extreme Programming," Amr Elssamadisy and Gregory Schalliol of ThoughtWorks describe a case where after 3 years of success with applying XP to a lease management system, the length of time to add a new feature became longer than an iteration due primarily to increasingly complex integration and technical debt issues.
Poor contracting	Development lopped off by a fixed-price, fixed SOW contract	TRW spent money and schedule designing a system to a 1-second response time requirement only to find that this was not affordable. Luckily, this was discovered early and only cost 13 months of schedule.
Policy influences (on standards development)	Stretched requirements result in wasteful expenditures on non-value adding work that stretch schedules and budgets	The definition of MIL-STD-498 as a replacement for 2167 and 7935. Wanting to avoid imposing 23 DIDs that on simple could be tailored down but in principle rarely were in practice, two other versions (one with 6 DIDs and one with 1 DID) were developed. The policy police decided that DoD couldn't have more than one set of documents covering the same content, leaving only the 23 DID version.
Policy influences (Expert-developed standards)	Lack of understanding, "short" sighted policy definition and "long" impacts leading to disastrous process implementations	The framers of 2167 and 2167A didn't see the waterfall diagrams as a problem, because "anyone with common sense would know better than to commit to requirements without establishing their feasibility." But less-expert project managers would see" following the standard" as the safest thing for their careers, and end up getting into trouble.
Policy Influences (Piling On Constraints)	Rework and technical debt overload stretch schedules and budgets	Changing the rules mid-stream with inflexible processes is disastrous. An organization started with the 2167 mandate to have the requirements determine the delivered capabilities. mid-way through the project, a SecDef memo to "use COTS products wherever possible," meant COTS capabilities would determine the requirements. Later in the development, a mandate to use the Ada programming language resulted in significant effort because many of the selected COTS products had weak or no Ada bindings.
Top-Executive Mandates	Unintentionally imposed constraints that cut off technical solution options	Dated executive experience often constrains their decisions. A 2006 Mark Maier SysE Journal paper identified hardware architecture constraints imposed by hardware-oriented top executives in terms of functional hierarchies and simple interfaces that cut off software options such as layered-service architectures and more complex but necessary interface protocol compatibility standards options.
Voice of the Customer.	Every customer need becomes a project requirement, stretching the project well outside budget and schedule constraints	The Bank of America Master Net project used a broad, unmediated Voice of the Customer approach that ended up in a disaster when the major stakeholders' agreed-to desires resulted in significant success model clashes and overruns.
Test-Driven Acceptance.	Under-constrained acceptability, leaving extremities to be lopped off later	The 3000-test Ada compiler validation suite led compiler vendors to patch their compiler software to pass the tests, creating a product that was often less robust than their beta-test versions.
Search-Driven Acceptance.	Projects deploying inappropriate practices, methods or approaches	Search engine results on the use of formal methods found mostly success stories, but on small projects, leading some projects to adopt the methods only discover scalability shortfalls.
Auditor-Driven Acceptance.	Varying auditor interpretations over constrain or under constrain projects leading to stretching or chopping later	Software CMM or CMMI auditor-based maturity levels requirements had little impact on acquisition programs.
Value-Neutral Acceptance.	Inappropriate activity and gaming on the part of developers driven by simplistic or incomplete metrics	Some projects use delivered defect density as the basis for acceptance, leading project personnel to fix the easy defects. The project then finds the hard defects are unacceptable, and must be stretched well beyond its budget to become acceptable.
Acquisition-Oriented Acceptance.	Product too expensive to operate and maintain	Tight budgets and schedules lop off options to design and develop the project to facilitate maintenance, operations, and support.

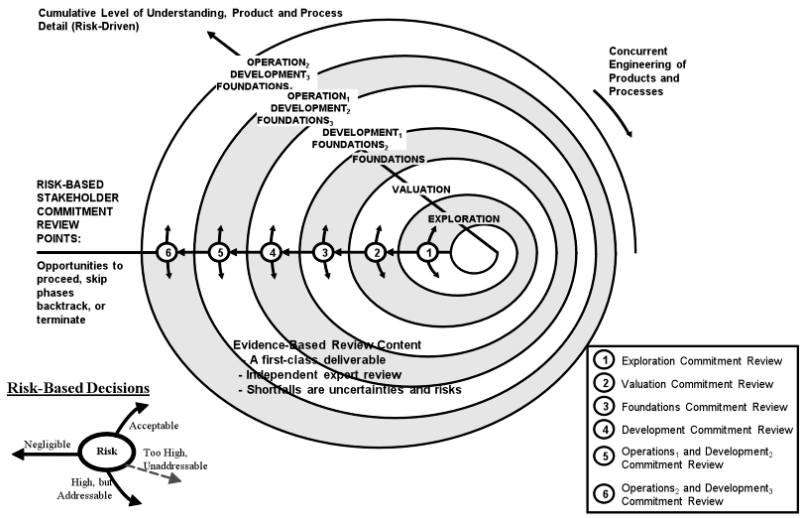


Figure 1. The Incremental Commitment Model: Spiral View

ICSM Lifecycle

The Phased View (Figure 2) shows how the overall lifecycle process divides naturally into two major stages. Stage I, Incremental Definition, covers the up-front growth in system understanding, definition, feasibility assurance, and stakeholder commitment. If the Phase I activities do not result in deciding to radically change the effort by adjusting scope or priorities, or discontinuing the development completely, they lead to a larger Stage II commitment to implement a feasible set of specifications and plans for Incremental Development and Operations.

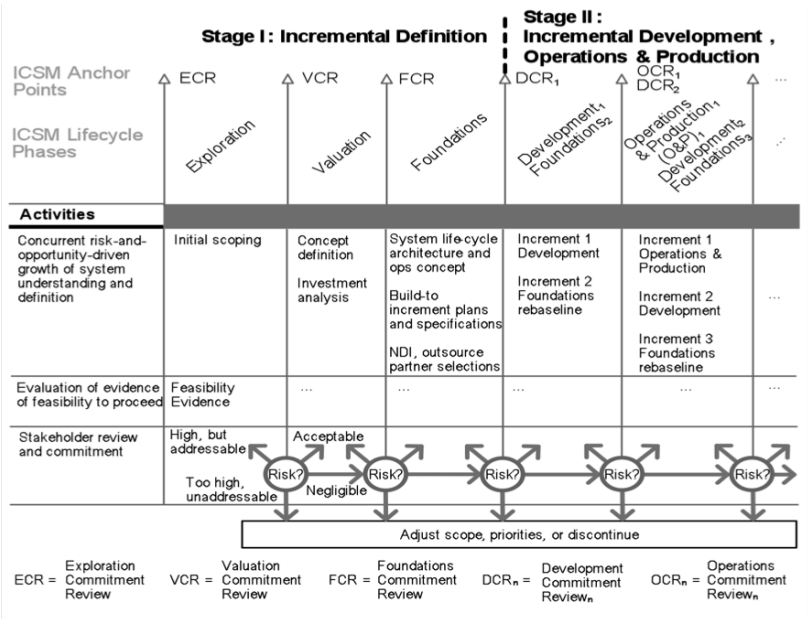


Figure 2. The ICSM Staged View

The duration of Stage I can be anywhere from one week to five years, depending on factors like the number, capability, and compatibility of the proposed system's components and stakeholders. A small, experienced, developer-customer team, using agile software methods and operating on a mature infrastructure, can form and begin incremental development of a well-defined software project in less than a week. A more complex project requires significant effort and could take up to five years or more. An example might be an ultra-large, unprecedented, multi-mission, multi-owner, system-of-systems needing to integrate with numerous independently evolving legacy or external systems. We have provided ICSM elements to the definition and development of such systems.[5]

Stage II is planned around the length of the increments to be used in the system's development and evolution. This is a key decision made during the Development Commitment Review. A small agile project can use two-to-four week increments. A much larger project could need increments of up to two years to develop and integrate an increment of operational capability. However, the ICSM capability delivery cadence is not necessarily linked to the internal development cadence, and there may be several internal integration cycles within a longer release increment. Some large, inseparable, hardware components would take even longer to develop their initial increments, and would be scheduled to synchronize their capability deliveries with concurrently evolving infrastructure or software increments.

Stage I activities have assured a common vision, committed stakeholders, and an architecture capable of accommodating foreseeable changes such as user interfaces, external system interoperability requirements, or transaction formats. These enable the features in each Stage II increment to be prioritized and the increment timeboxed.

Flexible, Multiple and Evolving Processes

The ICSM essentially uses evidence and risks to generate appropriate processes throughout the lifecycle. Figure 3 illustrates four example paths through the ICSM to visualize how different risks create different processes.

Example A is a simple business application based on an already-available Enterprise Resource Planning (ERP) package. There is no need for a Valuation or Architecting activity if the ERP package has already been purchased and its architecture has already proved cost-effective in supporting more complex applications. Thus, the project can go directly into Stage II, using an agile method such as a combination of Scrum and Extreme Programming. There is no need for "Big Design Up Front" activities or artifacts because an appropriate architecture is already present in the ERP package. Nor is there a need for heavyweight waterfall or V-model specifications and document reviews. The critical risk identified at the end of Exploration could be the user acceptance and business process reengineering required for deployment. In this case, that risk would be considered negligible if the system's human interface risks have been sufficiently mitigated via ERP package-based prototyping.

Example B involves a risky but innovative system such as adding a retina scanner to the next model of a cellphone product. There are a number of uncertainties and risks/opportunities to resolve, such as scanner hardware integration and safety of the user. But the new capability is needed quickly and there is a fallback (deferring its introduction to the following model), so proceeding to address the risks and develop the system is acceptable.

Example C is a system that is defined as safety critical. The stakeholders responsible for the safety of the proposed system find at the Foundations Commitment Review that the proposers have provided inadequate safety evidence. It is better to have the

proposers develop such evidence through architecture-based safety cases, fault tree analyses, and failure modes and effects analyses before proceeding into the Foundations phase. The arrow back into the Valuation phase indicates this.

In Example D, the developers are simply too late to play. It is discovered before entering the Development phase that a superior product has already entered the marketplace, leaving the current product with an infeasible business case. Here, unless adjusting the project's scope can make a viable business case, it is best to discontinue it. It is worth pointing out that it is not necessary to proceed to the next major milestone before terminating a clearly non-viable project; however, stakeholder concurrence in termination is essential.

ICSM Risk-Driven Common Cases

Many projects can reuse experience from previous projects. However, every project has the possibility of unique aspects that could impact the selection of processes and the path through the ICSM. To enable early estimation, supply examples that help users with initial planning, and support categorization and capture of lessons learned, we have identified a set of seven risk patterns that represent the most often seen paths through the ICSM. We have named these patterns Common Cases:

- Software application or system
- Software-intensive device
- Hardware platform
- Family of systems or product line
- System of systems (SoS) or enterprise-wide system
- Brownfield modernization

Table 2 briefly describes when to use each common case and some examples of each.

ICSM and Large, Complex Systems

Obviously, larger, more complex systems will require a great deal more activity in Stage I. In Stage II, however, the ICSM allows a great deal of flexibility in providing a way of integrating and accommodating the wide variety of development activities that can appear across the various hardware, software, and human development activities. For that reason, the Implementation Phase is based on a three-tiered, timeboxed process that allows for reflection, anticipation, and adjustment to the changing environment, shown in Figure 4. This concept works best in software, but can apply to hardware in many cases. Figure 5 shows how this three-tiered model scales to multiple component or subsystem development.

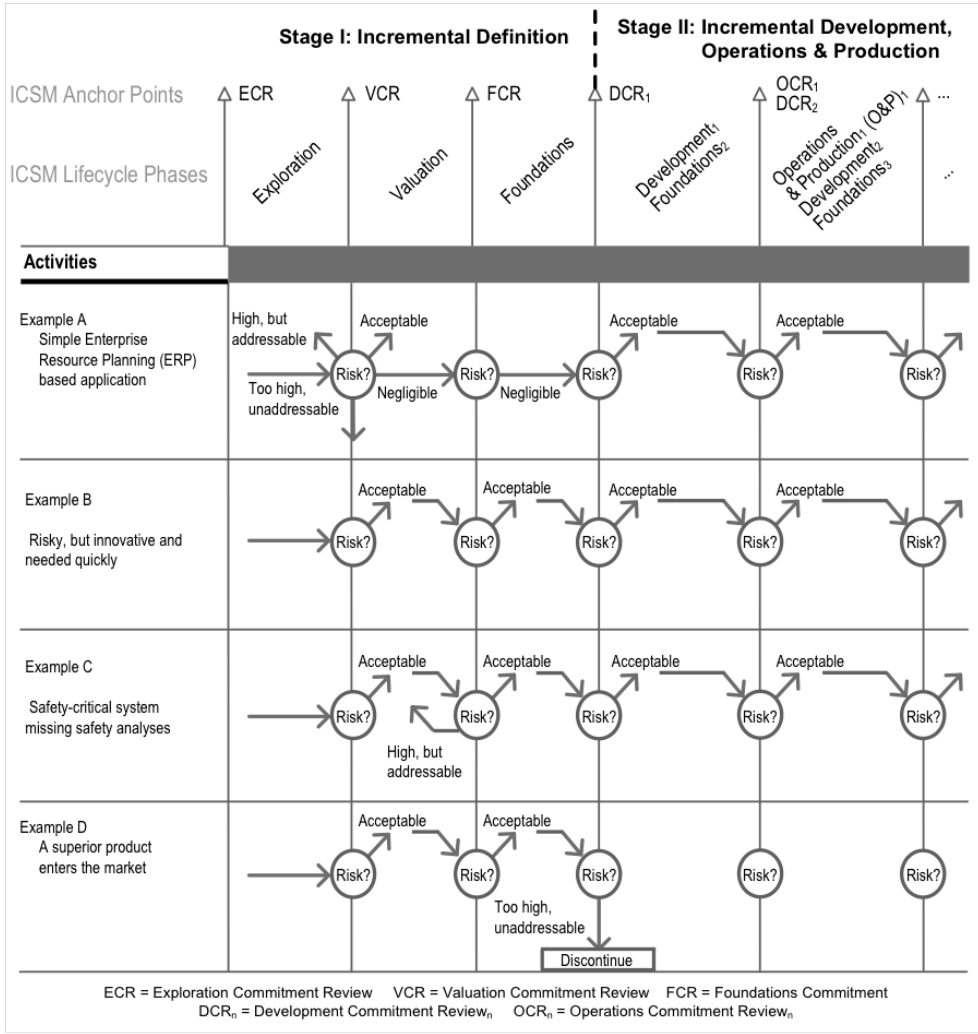


Figure 3. Different Risk Patterns Yield Different Processes

System (or subsystem)	Common Case	Examples
SW application/system executing on one or more commercial HW platforms, as a standalone system or a constituent of one or more SoSs.	SW application or system	Cellphone app, business application or system, military command and control software system, inventory management systems, computer operating system, database management system
A special purpose object, machine, or piece of equipment that has significant features provided by software.	SW-intensive device	Computer peripherals, weapons, entertainment devices, health care devices (including small surgical), GPS receivers, manufacturing tools
Vehicle (land, sea, air, or space)	HW platform	Small unmanned vehicle, automobile, tank, ship, airplane, space shuttle, space station, Mars rover
Computer	HW platform	Mainframe, server, laptop, tablet, cellphone
Part of a set of systems that are either similar to each other or interoperate with each other	Family of systems or product line	Car models that share many core components; interoperating back-office systems such as billing, accounting, and sales force support, that share a common repository with standard data definitions and formats, and are provided by a single vendor
A new capability that will be performed by more than one interoperating system	SoS or enterprise-wide system	Multiple interoperating systems owned and managed by different organizations; for example, navigation systems that include airborne and land systems using GPS
Refactoring or re-implementation of an older legacy system or set of systems	Brownfield modernization	Incremental replacement of old, fragile business systems with COTS products or technology refresh/upgrade of existing systems

Table 2. ICSM Common Cases

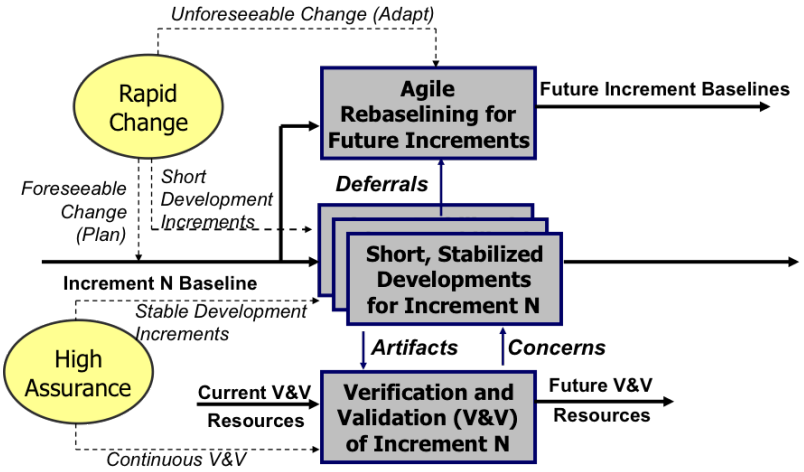


Figure 4. Three-tier timeboxed approach (Evolution View)

ICSM and Process Improvement

ICSM is designed to provide flexibility. It also expects you to evaluate and apply the process assets you already have in new ways, and provides essential guidance on hw that can happen. ICSM also seeks to actively create and use lessons learned both within and between projects to decrease the learning cycle and accelerate improvement. The key intrinsic process improvement aspects in ICSM are evidence, risk-based process, the incremental approach, and anticipation/reflection.

In the ICSM, evidence is continuously created as a first class deliverable and used for process generation, decision-making, and stakeholder commitment. This evidence captures a wide variety of knowledge in a way that can be empirically analyzed to support retrospection at almost every point in the lifecycle. It can also be used to improve estimation, evaluate experimental processes and methods, and transfer knowledge across projects and systems.

As with many process models, risks are captured and tracked. However, in the ICSM they also directly impact the process generation activities and are integrated into all decision-making. Many risks are common across a domain, and so mitigation efforts based on ICSM process decisions are documented and can be easily captured to support decision-making and process generation across projects.

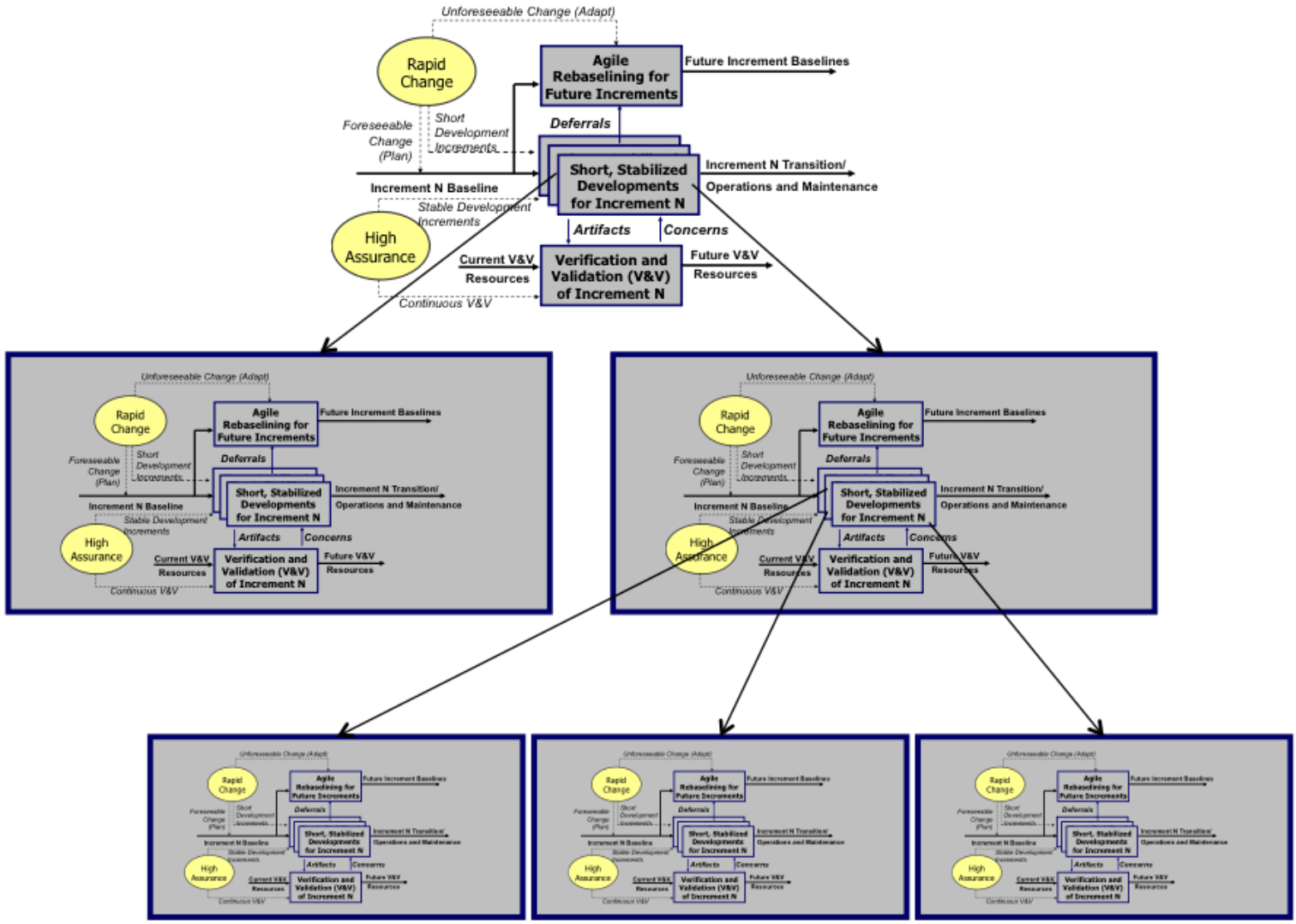


Figure 5. A Large-system development phase

Issue	ICSM Mitigation
Defined process mismatch	Track evidence of time needed to both develop and integrate new increments, and adjust increment sizes and/or schedules as necessary
Poor contracting	Develop evidence of the need for 1-second response time and the cost of achieving it before committing to it.
Policy influence (on standards development)	Develop and sustain multiple sources of guidance for deliverables on different classes of systems, such as with the recent draft update of DoDI 5000.02
Policy influence (Expert-driven standards)	Provide criteria for initial choice of project process, and risk-based decision guidance on process adaptation to change
Policy Influence (Piling On Constraints)	Add new guidance directives only based on evidence of their compatibility with existing directives
Top-executive Mandates	Involve development and support stakeholders in key process and product guidance. Concurrently engineer the system's hardware, software, and human elements
Voice of the Customer.	Involve all success-critical stakeholders in key project and product guidance decisions
Test-driven Acceptance	Evolve test criteria based on user alpha, beta-test experience
Search-driven Acceptance	Ensure that evidence is accumulated from fully representative stakeholder communities
Auditor-driven Acceptance	Involve stakeholders in choice of process and product guidance.
Value-neutral Acceptance	Use stakeholder value propositions to prioritize requirements, proposed changes, test cases, defect fixes
Acquisition-oriented Acceptance	Involve post-deployment stakeholders in determination and prioritization of requirements

Table 3.ICSM mitigations to procrustean issue

The incremental nature of the ICSM shortens the learning cycle. Agile and lean development methods with short cycle times, value-based scheduling, and continuous integration can be employed wherever appropriate. Coupled with the ICSM emphasis on evidence and risk, these can accelerate learning, reduce rework, and manage technical debt in such a way as to provide continuous process improvement throughout the Stage II activities.

Finally, process improvement requires balanced reflection and anticipation. Wayne Gretzky, who is generally acknowledged as the greatest hockey player of all time, ascribes a good deal of his success to the ability to anticipate where the hockey puck was going, and to skate to where he could capitalize on that knowledge. Anticipating where technologies, competitors, organizations, and the marketplace are going is increasingly critical to successful systems and software engineering. In contrast, organizations that spend their time asking, "How could we have done our last project better?" are actually skating to where the puck has been. Clearly, such "reflection in action" is good,[6] but in a world of rapid change, reflection in action needs to be balanced with anticipation. The Incremental Commitment Spiral Model integrates reflection, anticipation, and agility to take advantage of evolving knowledge through a risk-based, principle-driven approach to system development. We are still firm believers that there are no panaceas, silver bullets, or one-size-fits-all solutions. We are confident, though, that the ICSM offers a coherent and useful way to approach systems development in a world that has not only changed, but will also continue to change throughout every system's life cycle.

Conclusions

Procrustes caused a lot of damage before Theseus turned the tables (or the bed) on him. We believe that there are a lot of ways to fight procrustean tendencies through rethinking the processes we advocate, and pushing back on those who are applying inappropriate or damaging processes to our projects. One of these ways is using the process generation framework provided by the ICSM. Table 3 shows how the ICSM can mitigate our earlier list of procrustean issues.

ISCM supports adapting and applying multiple processes (or process assets) as needed throughout a project, regardless of size, duration, or complexity. It provides a flexible, extensible lifecycle that can be adopted across a wide variety of project environments. Most importantly, it establishes all of the underlying principles of high maturity organizations— stakeholder value, incrementality, concurrency, agility, flexibility, empiricism, improvement and predictability—without restricting the specific processes deployed. ICSM enables the opposite of a procrustean process: one that adapts to your needs rather than forcing you to meet its own.

ABOUT THE AUTHORS

Dr. Barry Boehm is a USC Distinguished Professor and Chief Scientist of the DoD-Stevens-USC Systems Engineering Research Center,. He was director of DARPA-ISTO 1989-92, at TRW 1973-89, at Rand Corporation 1959-73, and at General Dynamics 1955-59. He is a Fellow of the primary professional societies in computing (ACM), aerospace (AIAA), electronics (IEEE), and systems engineering (INCOSE), and a member of the U.S. National Academy of Engineering.

E-mail: barryboehm@gmail.com

Dr. Richard Turner is a Distinguished Service Professor at the Stevens Institute of Technology. Active in the agile, lean and kanban communities, he helped author the Software Extension to the PMI Guide to the PMBOK. He is a Golden Core member of the IEEE Computer Society, a fellow of the Lean Systems Society and co-author of four books: The Incremental Commitment Spiral Model, Balancing Agility and Discipline, CMMI Survival Guide, and CMMI Distilled.

Phone: 202-390-3772
E-mail: rturner@stevens.edu

Jo Ann Lane is currently the systems engineering Co-Director of the University of Southern California Center for Systems and Software Engineering, a member of the Systems Engineering Research Center Research Council representing the system of systems research area, and emeritus professor of computer science at San Diego State University. Her current areas of research include system of systems engineering, system affordability, expediting systems engineering, and balancing agile techniques with technical debt.

Phone: 858-945-0099
E-mail: jolane@usc.edu

Dr. Supannika Koolmanojwong is a lecturer and a researcher at the University of Southern California Center for Systems and Software Engineering. Her primary research areas are Software Process Improvement, Software Process Quality Assurance, Software Metrics and Measurement, Agile and Lean Software Development and Expediting Systems Engineering. She is a certified scrum master and a certified Product Owner. Prior to this, she was a software engineer and a RUP/OpenUp Content Developer at IBM Software Group.

E-mail: koolmano@usc.edu

NOTES

1. Boehm, B. and J. Lane, "Using the Incremental Commitment Model to Integrate System Acquisition, Systems Engineering, and Software Engineering," CrossTalk, October, 2007
2. Boehm, B., J. Lane, S. Koolmanojwong, and R. Turner, The Incremental Commitment Spiral Model: Principles and Practices for Successful Systems and Software, Addison Wesley Pearson, New York, 2014.
3. Boehm, B. "A Spiral Model for Software Development and Enhancement." Computer. May 1988;61–72.
4. CrossTalk. "Top Five Quality Software Projects." January 2002, July 2003, July 2004, September 2005. www.stsc.hill.af.mil/crosstalk.
5. Stephen Blanchette Jr., Steven Crosson, Barry Boehm, "Evaluating the Software Design of a Complex System of Systems," CMU/SEI Tech Report CMU/SEI-2009-TR-023, January 2010
6. D. Schon, The Reflective Practitioner. Basic Books, 1983.

REFERENCES

1. Much of the material in this article is drawn from a new book: Boehm, B., J. Lane, S. Koolmanojwong, and R. Turner, The Incremental Commitment Spiral Model: Principles and Practices for Successful Systems and Software, Addison Wesley Pearson, New York, 2014. The initial work that provided the basis for the book was funded in part by the US Department of Defense, through the Systems Engineering Research Center, a University Affiliated Research Center at Stevens Institute of Technology.

Disciplined Learning
The Successor to Risk Management

Alistair Cockburn, Humans and Technology

Abstract. Disciplined learning, or "learn early, learn often," updates naïve agile development and traditional risk management, and safely replaces the dreaded catch phrase, "fail early fail often." Disciplined learning is a rich, creative and rewarding endeavor, already in use in small pockets of excellence.

Introduction

Naïve agile development works remarkably well, given how simple it is. It is less than optimal, however, and insufficient for many situations. Disciplined learning adds to agile.

Traditional risk-management generally addresses how to avoid failure rather than how deliver success. Disciplined learning updates risk management by incorporating some of the principles of agile development.

Disciplined learning is neither obvious nor for the faint of heart, but it is in active use by top teams in many disciplines, who manage to deliver success in difficult circumstances.

Consider, as a reference point, the still-common way of working in which a major integration or delivery occurs at the end of a long period of work without integration or delivery (see Figure 1). It is not necessary to be working in a waterfall fashion to have this moment of integration or delivery in the project, so the curve need not be ascribed to waterfall. It is a simply a common strategy.

Figure 1 shows time on the horizontal axis. The dotted line shows project costs increasing steadily over time. The solid line shows that learning progresses while the project teams work, talk, design, but not in the major way that learning (and surprises) occur immediately after the moment of integration or delivery.

Learning occurs relatively late in the project, after most of the cost has been accrued.

What we are after is how to learn earlier in the project, when changes can be made with lower cost. This is where creativity and discipline come in.

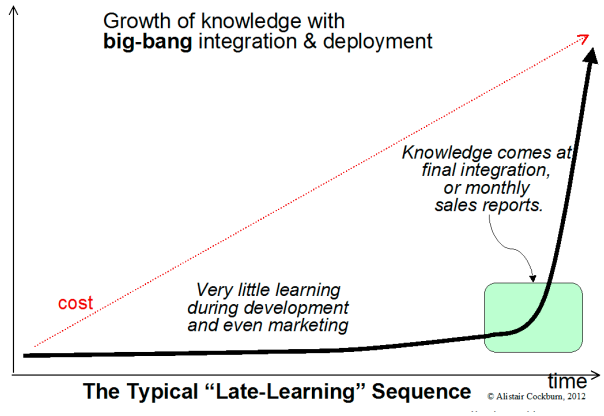


Figure 1. The typical "late-learning' strategy.

Four Learning Topics

The team has (at least) four categories in which to learn:

- What they should really be building, never mind what they thought they should build at the start.
- Whether they have the right people on the team, and for those people, how best to work together.
- Where their technical ideas are flawed.
- How much it will cost to develop.

In the strategy shown in Figure 1, these are all learned late in the project, around the time when the parts are integrated and deployed, when the consumers finally give feedback on the result. This learning arrives too late to benefit the product.

The disciplined learning approach is to apply the same "broken" learning curve in very small doses, deliberately and often, so that each step provides information that can be used to adjust the four categories of learning. The payoff is not just reduced risk in the final delivery, but the ability of the sponsors to steer the final delivery in a fine-grained way, both in delivery time and delivered features and quality.

Figure 2 illustrates the disciplined learning approach. The following four sections describe strategies for learning in the four categories.

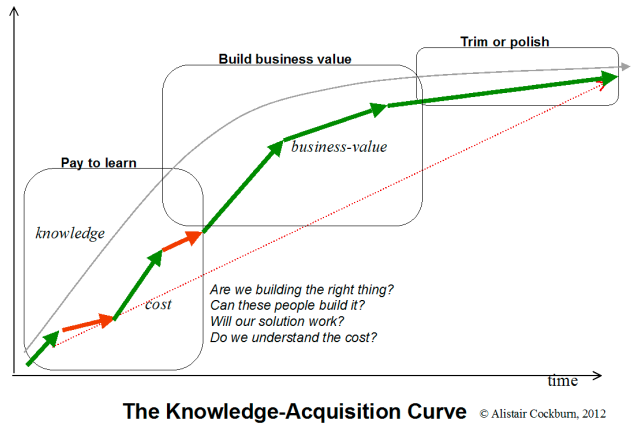


Figure 2 Applying the principle: Learn Early, Learn Often.

Learn What Should Get Built

The most important and most difficult question is: Will people like, buy and use what we're building?

Normally, this question gets answered when it is too late. Recently, however, strategies have come into usage that move this learning process forward. The strategies are fairly simple, but require discipline, patience, and a willingness to change course based on the results.

Sample strategies are:

- Paper prototyping.
- Ambassador user.
- Early delivery.
- Empty or manual delivery.

Paper prototyping [1] and related strategies coming from the user-centered design community [2] involve nothing more com-

plicated than putting a mockup of the product into the hands of the consumer, who reacts to these early design thoughts. Prepared at low cost, early in the development cycle, these prototypes allow the development team to change their minds about how to proceed.

An "ambassador user" is a friendly user to whom the team can deliver an incomplete but growing product. This user usually breaks the system within moments, and give valuable feedback from his or her (limited) perspective. The difference between the "ambassador user" and "paper prototyping" is that the ambassador user is encountering the actual system as it grows, not a mockup of the system.

"Early delivery" is a full deployment of the system with reduced capabilities. The intention is to learn, first of all, what is incorrect with the product as envisioned, but possibly more significantly, how the presence of the product changes the thoughts about what should be built in the first place. "Early delivery" recognizes that once people start using a system, their habits and needs change, often in unpredictable ways. Delivering a thin version of the system early allows the development team to gather new input and adjust the priorities on what should be developed.

The above are all standard albeit frequently ignored techniques, found in the regular and agile literature.

The most interesting strategies to emerge in the last decade are two documented and practiced in the lean startup community: Empty and Manual delivery (my terms for them).

The "Empty Delivery" [3] strategy is particularly well suited for online products. Initially, all that is detected is whether anyone clicks on a link or accesses a feature. There is no implementation behind the façade of the click. Measuring these clicks, a team can reduce or sequence the features developed to follow those drawing the most attention. The system evolves in the direction of maximum draw.

"Manual Delivery" is described in Eric Ries' book, The Lean Startup [4]. In this strategy, a team spends what may seem to be excessive money even delivering products manually, for the simple reason that manual procedures can be set up and changed for very little cost. Delivering manually, the team can change the product offering with every single purchase, evolving to what the customer base indicates is really desired.

Adjust Design Decisions

Mistakes in design come from:

- Choosing technology that doesn't work as advertised.
- Mistakes due to people not talking to each other, with resultant mistaken assumptions about each other's work.
- Inevitable omissions and mistakes in design.

These mistakes are discovered and repaired using strategies:

- Walking skeleton.
- Micro-incremental development.
- Spikes.
- Story splitting.

The "Walking Skeleton" strategy [5] calls for the team to connect a thin path through the architecture. In creating this simple but full system, they discover the first round of surprises in the technologies they are using.

Once the system is thinly connected, the infrastructure and functionality teams each adds onto their part of the system. It is not uncommon to see the infrastructure team redesigning the skeleton itself, while keeping the interfaces to the functionality running (or forcing updates). This restructuring is one of the costs of using the strategy.

Micro-incremental development is when teams integrate their work every hour, half-day, or day. The shorter the time between integrations, the faster they find mistakes, and the lower the cost of making changes. A side benefit is that they are less likely to change the same part of the design at the same time, and so they do not need to check out and branch the design, making integration easier, faster, and less error prone.

A spike [6,7] is a small, disposable piece of work created to explicitly address the question, "Is there an obvious flaw in this approach?" It is used to flush out interface mismatches as well as various performance and scaling problems.

The difference between a spike and ordinary incremental development is that ordinary incremental development is conducted using full production conventions, with the assumption that the work will be used in the final product. A spikes must absolutely not be used in the final product; it is throwaway work. Because the work is throwaway, it is always done in the most rapid and effective manner possible with the sole purpose of learning about the question at hand.

Some questions might seem impossible to move forward in the schedule, such as the final conversion of the database. With story splitting [8] a story is split into a learning (spike) piece and a production piece. The spike is placed early to learn how to address whatever difficulties might lie in it. Then the actual work can be left until the appropriate moment in the schedule.

Learn to Work Together

Failure to deliver is sometimes due not to the people being not correct for the assignment, but to them not having learned how to work together. Tom DeMarco and Tim Lister refer to a "jelled team" [9]. Three strategies help with creating a jelled team:

- Early victory.
- Walking skeleton.
- Simplest first, worst second.

The Early Victory [10] strategy is based on the work of sociologist Karl Weick [11], showing that achieving results helps people come to trust each other more, raises morale and helps them perform better.

The "walking skeleton" already described produces an early technical victory to the team and to the sponsors. The concept is sometimes adjusted to implement and deliver a thin path through the workflow of a company, with similar "early victory" and technical learning for the delivery and work flow aspects of the project.

The "simplest-first, worst second" strategy [12] is contrary to the usual recommendation in the agile development world. The usual agile advice is to build the highest business value first. That strategy makes good sense once the team is functioning well, social risks have been reduced, and the team is capable

and confident of being able to deliver whatever is of the highest business value. However, many conversations need to take place before the team has reached that point. For this reason, it is sometime useful to build something real but very simple, so that they can adjust social habits in good time before the difficult parts of the project are reached.

Learn How Much It Will Cost

Two strategies help with learning the cost of a project:

- Core samples.
- Microcosm.

Tim Lister told the following story at a conference [13], "A man wanting a pool built in his back yard calls in three contractors to present estimates. The third contractor, instead of presenting an estimate, tells the homeowner he will need to drill and core sample in the ground, and will charge the man for that. The homeowner complains, saying that the first two contractors didn't charge him for core sampling. The contractor responds that he has no idea how the first two contractors could submit a bid, since they don't know what sorts of rock layer lies under the lawn, but he couldn't possibly put in a bid without having that information. The homeowner now comfortable with the third contractor, hires him for the work."

To do this with a development project, isolate parts of the system the development of which is not obvious and develop very small elements within those areas. In that development, identify what sorts of surprises lurk below the surface and understand how difficult the work will really be. Carefully selecting such "core samples" allows the team to develop a more reliable cost-, time-, and resource estimate for the project.

Core sampling is the miniature version of the more general "Microcosm" strategy [14], in which a mini-project is run for the sole purpose of establishing a sound estimate. A full Microcosm project can be set up to test the productivity of a new development team (think off-shoring, in particular), as well as to test the learning speed of staff with new technologies, to benchmark the productivity of expert versus ordinary or new developers.

Whereas a core sample effort is intended to take hours to days, a full Microcosm project may take weeks to carry out, and should therefore only be used for larger development efforts.

Creating a Plan

In the light of these strategies, the creation of a project plan is rather different than before.

Disciplined learning calls for merging learning steps from the four categories above with requests for growth of business value as is standard with incremental development. Business value and learning are artfully interleaved into a sequence of work assignments designed to reduce risk, deliver crucial information, and develop product capability in an "optimal" way.

This is where creativity enters.

The quality of the plan is sensitive to the ability of the planners to identify and merge the learning needs and the upcoming possibilities for income. As lessons are learned and new risks and opportunities spotted, the project will need to be updated.

Trimming the Tail

A product feature actually consists of three parts, not just the two:

- Learning.
- Value.
- Tail.

The "tail" is the polishing and glossing that makes a feature "wonderful." Since not every feature is of equal value to the buyers and users, many or even most features can be thinned or trimmed back without damage to the system.

Attending to the presence of a tail, a team can arrange for a minimum set of features to be at an "adequate" level of wonderfulness in plenty of time before final delivery, then spend the remaining time polishing and glossing those feature that are more important than the others [15]. Alternatively, if time is short, they can cut back on (trim) the polishing and deliver early or on time [16]. This is described in the final section.

Reaping the Benefits

Disciplined learning delivers two benefits: early income and the ability to trim the tail.

Early income from incremental development is well presented in Software by Numbers [17]. A project can become self-funding if it is delivered to paying users part-way through its development, thus lowering the load on the sponsors.

Less obvious but equally valuable is the ability to not develop less valuable aspects of the system. Here is the shortest example, to give the idea:

When you are opening a new hotel, it may not be necessary to shine the doorknobs before opening to the public. If it is necessary to have shined doorknobs for the guests, it is probably not necessary that all of the doorknobs need be shined.

You might trim any of four aspects of a system:

- Features.
- Feature details.
- Usage quality.
- Internal quality.

You drop an entire feature. A car (for example) might not need a sunroof. The first iPads did not have phone modems.

If not an entire feature, you might be able to trim an aspect of a feature: Given that your car must have all of the basics (such as brakes), it might not need brakes with antilock braking. A computer system might require searching capability, but not auto-completion or auto-correction.

Recognizing that really smooth and easy to use features take a lot of work, you might choose to skip improving usability for selected features.

Finally, you can trim internal design quality and correctness. The question is how much internal quality is needed for the delivery in question.

If development has proceeded incrementally, attending to the learning areas, then the team can deliver:

- Early, with reduced features or quality.
- On time, with either full or reduced quality, depending on where development stands at that time.
- Or later, with enriched features or quality; at the choice of the sponsors!

Under usual project circumstances, the only choices are to delay or work overtime. The "trim the tail" option is available only for those who have worked in this more disciplined fashion.

Disciplined learning with trim-the-tail is one of the few approaches equally available to very small and very large projects, fixed-price and floating-price projects. Here are three examples, taken from real projects:

1. Small, floating-price project: A web site development involving only the web site owner and the programmer. After several months of open-ended work, the web site owner wanted the site delivered "soon," and trimmed the tail back aggressively and repeatedly until something much smaller than expected but still suitable was deployed.

2. Small, fixed-price project: The company in question always bid small, fixed-price contracts of three- to six-months, involving three to eight people. As usual, the bids were aggressive and the teams typically ended late, missing the deadline or scope, with resulting overtime from the developers and penalties at the end of the contract. Jeff Patton [18] worked in the manner described in this article, leaving the least important features to the end, and deliberately thinning the less critical features, so that when the contract period ended, it was clear to the customers that they had gotten most of what they wanted. This produced the least overtime, the smallest penalties, the highest customer satisfaction and the greatest likelihood of receiving a follow-on contract.

3. Very large development project: A company with several thousand developers in several countries, working on a product line with multiple variations, applications and releases. Under normal circumstances, when they call for a full integration on a particular date, every team starts to work overtime and jockey for position not to be the one most behind schedule. The integration date keeps getting slipped back as team after team fails to complete their work on time. Using the trim-the-tail approach, each team would have in place the essential elements needed for the integration, with only tail elements left unfinished. For delivery, management would be in position to deliver slightly less, on time, or slightly more, a bit later.

It is exciting to find a baseline strategy that applies to projects of such different sizes and natures as just outlined.

Disciplined learning is not for the faint of heart. It requires discipline, creativity and constant correction. The payoff is the ability to get a team working together, discover what is needed in time, deliver it early in order to create a self-funding project, and finally, trim the tail at the end to meet inelastic deadlines.

ABOUT THE AUTHOR

Dr. Alistair Cockburn, one of the creators of the Manifesto for Agile Software Development, was voted one of the "The All-Time Top 150 i-Technology Heroes" in 2007 for his pioneering work in use cases and agile software development. An renowned IT strategist and author of the Jolt award-winning books "Agile Software Development" and "Writing Effective Use Cases," he is an expert on agile development, use cases, process design, project management, and object-oriented design. In 2001 he co-authored the Agile Manifesto, in 2003 he created the Agile Development Conference, in 2005 he co-founded the Agile Project Leadership Network, in 2010 he co-founded the International Consortium for Agile. Many of his articles, talks, poems and blog are online at <http://alistair.cockburn.us>.

E-mail: totheralistair@aol.com

REFERENCES

1. <http://en.wikipedia.org/wiki/Paper_prototyping>
2. <http://en.wikipedia.org/wiki/User-centered_design>
3. BBC "Searching the internet's long tail and finding parrot cages," <http://www.bbc.co.uk/news/business-11495839>
4. Reis, E., The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses, Crown Business, 2011.
5. <http://alistair.cockburn.us/Walking+skeleton>
6. <http://c2.com/xp/SpikeSolution.html>
7. <http://agiledictionary.com/209/spike/>
8. <http://alistair.cockburn.us/The+A-B+work+split>
9. Tom DeMarco and Timothy Lister, Peopleware: Productive Projects and Teams. New York: Dorset House Publishing Co., 1987.
10. <http://alistair.cockburn.us/Advancedpmstrategies1-180.ppt>
11. Karl Weick, The Social Psychology of Organizing, McGraw-Hill Humanities/Social Sciences/Languages; 2nd edition, 1979.
12. Alistair Cockburn, Crystal Clear: A Human-Powered Methodology for Small Teams, Addison-Wesley, 2005. Also online at <http://alistair.cockburn.us/ASD+book+extract%3A+%22Individuals%22>
13. Lister, Tim, keynote at Agile Development Conference 2010.
14. <http://alistair.cockburn.us/Project+risk+reduction+patterns>
15. <http://www.agileproductdesign.com/downloads/patton_embrace_uncertainty_optimized.ppt>
16. <http://alistair.cockburn.us/Trim+the+Tail>
17. Mark Denne and Jane Cleland-Huang. Software by Numbers: Low-Risk, High-Return Development. Prentice-Hall, 2003.
18. Jeff Patton, "Unfixing the Fixed Scope Project: Using Agile Methodologies to Create Flexibility in Project Scope," in Agile Development Conference 2003, Proceedings of the Conference on Agile Development, 2003, ACM Press. Available online through a Google docs search.

Achieving Software Excellence

Capers Jones, Namcook Analytics LLC

Abstract. In 2014 software is the main operational component of every major business and government organization in the world. But software quality is still not acceptable for many applications. Software schedules and costs are frequently much larger than planned.

This short study discusses the proven methods and results for achieving software excellence. The paper also provides quantification of what the term "excellence" means for both quality and productivity.

Introduction

Software is the main operating tool of business and government in 2014. But up through the end of 2013 software quality remained marginal; software schedules and costs remained much larger than desirable or planned. Cancelled projects were about 35% in the 10,000 function point size range and about 5% of software outsource agreements ended up in court in litigation. This short study identifies the major methods for bringing software under control and achieving excellent results.

The first topic of importance is to show the quantitative differences between excellent, average, and poor software projects in quantified form. Table 1 shows the essential differences between software excellence and unacceptable results for a mid-sized project of 1,000 function points or about 53,000 Java statements.

The data in table 1 comes from the author's clients, which consist of about 600 companies of whom 150 are Fortune 500 companies. About 40 government and military organizations are also clients, but table 1 is based on corporate results rather than government results. Government software tends to have large overhead costs and extensive status reporting that are not found in the civilian sector. (Some big defense projects have produced so much paperwork that there were about 400 English words for every Ada statement, and the words cost more than the source code.)

(Note that the data in this report was produced using the Namcook Analytics Software Risk Master™ (SRM) tool. SRM can operate as an estimating tool prior to requirements or as a measurement tool after deployment.)

At this point it is useful to discuss and explain the main differences between the best, average, and poor results.

Topics	Excellent	Average	Poor
Monthly Costs			
(Salary+overhead)	$10,000	$10,000	$10,000
Size at Delivery			
Size in function points	1,000	1,000	1,000
Programming language	Java	Java	Java
Language Levels	6.25	6.00	5.75
Source statements per function point	51.20	53.33	55.65
Size in logical code statements	51,200	53,333	55,652
Size in KLOC	51.20	53.33	55.65
Certified reuse percent	20.00%	10.00%	5.00%
Quality			
Defect potentials	2,818	3,467	4,266
Defects per function point	2.82	3.47	4.27
Defects per KLOC	55.05	65.01	76.65
Defect removal efficiency (DRE)	99.00%	90.00%	83.00%
Delivered defects	28	347	725
High-severity defects	4	59	145
Security vulnerabilities	2	31	88
Delivered per function point	0.03	0.35	0.73
Delivered per KLOC	0.55	6.50	13.03
Key Quality Control Methods			
Formal estimates of defects	Yes	No	No
Formal inspections of deliverables	Yes	No	No
Static analysis of all code	Yes	Yes	No
Formal test case design	Yes	Yes	No
Testing by certified test personnel	Yes	No	No
Mathematical test case design	Yes	No	No
Project Parameter Results			
Schedule in calendar months	12.02	13.80	18.20
Technical staff + management	6.25	6.67	7.69
Effort in staff months	75.14	92.03	139.98
Effort in staff hours	9,919	12,147	18,477
Costs in Dollars	$751,415	$920,256	$1,399,770
Cost per function point	$751.42	$920.26	$1,399.77
Cost per KLOC	$14,676	$17,255	$25,152
Productivity Rates			
Function points per staff month	13.31	10.87	7.14
Work hours per function point	9.92	12.15	18.48
Lines of code per staff month	681	580	398
Cost Drivers			
Bug repairs	25.00%	40.00%	45.00%
Paper documents	20.00%	17.00%	20.00%
Code development	35.00%	18.00%	13.00%
Meetings	8.00%	13.00%	10.00%
Management	12.00%	12.00%	12.00%
Total	100.00%	100.00%	100.00%
Methods, Tools, Practices			
Development Methods	TSP/PSP	Agile	Waterfall
Requirements Methods	JAD	Embedded	Interview
CMMI Levels	5	3	1
Work hours per month	132	132	132
Unpaid overtime	0	0	0
Team experience	Experienced	Average	Inexperienced
Formal risk analysis	Yes	Yes	No
Formal quality analysis	Yes	No	No
Formal change control	Yes	Yes	No
Formal sizing of project	Yes	Yes	No
Formal reuse analysis	Yes	No	No
Parametric estimation tools	Yes	No	No
Inspections of key materials	Yes	No	No
Static analysis of all code	Yes	Yes	No
Formal test case design	Yes	No	No
Certified test personnel	Yes	No	No
Accurate status reporting	Yes	Yes	No
Accurate defect tracking	Yes	No	No
More than 15% certified reuse	Yes	Maybe	No
Low cyclomatic complexity	Yes	Maybe	No
Test coverage > 95%	Yes	Maybe	No

Table 1: Comparisons of Excellent, Average, and Poor Software Results

Software Quality Differences for Best, Average, and Poor Projects

Software quality is the major point of differentiation between excellent results, average results, and poor results.

While software executives demand high productivity and short schedules, the vast majority do not understand how to achieve them. Bypassing quality control does not speed projects up: it slows them down. The number 1 reason for

enormous schedule slips noted in breach of contract litigation where the author has been an expert witness is starting testing with so many bugs that test schedules are at least double their planned duration.

The major point of this article is: High quality using a synergistic combination of defect prevention, pre-test inspections and static analysis is fast and cheap. Poor quality is expensive, slow, and unfortunately far too common because most companies do not know how to achieve it. High quality does not come from testing alone. It requires defect prevention such as Joint Application Design or embedded users; pre-test inspections and static analysis; and formal test case development combined with certified test personnel.

The defect potential information in table 1 includes defects from five origins: requirements defects, design defects, code defects, document defects, and "bad fixes" or new defects accidentally included in defect repairs. The approximate distribution among these five sources is:

1. Requirements defects	**15%**
2. Design defects	**30%**
3. Code defects	**40%**
4. Document defects	**8%**
5. Bad fixes	**7%**
6. Total Defects	**100%**

However the distribution of defect origins varies widely based on the novelty of the application, the experience of the clients and the development team, the methodologies used, and programming languages. Certified reusable material also has an impact on software defect volumes and origins.

Because the costs of finding and fixing bugs have been the #1 cost driver for the entire software industry for more than 50 years, the most important difference between excellent and mediocre results are in the areas of defect prevention, pre-test defect removal, and testing.

All three examples are assumed to use the same set of test stages, including:

1. **Unit test**
2. **Function test**
3. **Regression test**
4. **Component test**
5. **Performance test**
6. **System test**
7. **Acceptance test**

The overall defect removal efficiency levels of these 7 test stages range from below 80% for the worst case up to about 90% for the best case.

Testing alone is not sufficient to top 95% in defect removal efficiency (DRE). Pre-test inspections and static analysis are needed to approach or exceed the 99% range of the best case.

Excellent Quality Control

Excellent projects have rigorous quality control methods that include formal estimation of quality before starting, full defect measurement and tracking during development, and a full suite of defect prevention, pre-test removal and test stages. The combination of low defect potentials and high defect removal efficiency (DRE) is what software excellence is all about.

Companies that are excellent in quality control are usually the companies that build complex physical devices such as computers, aircraft, embedded engine components, medical devices, and telephone switching systems. Without excellence in quality these physical devices will not operate successfully. Worse, failure can lead to litigation and even criminal charges. Therefore all companies that use software to control complex physical machinery tend to be excellent in software quality.

Examples of organizations with excellent software quality in alphabetical order include Advanced Bionics, Apple, AT&T, Boeing, Ford for engine controls, General Electric for jet engines, Hewlett Packard, IBM, Motorola, NASA, the Navy for weapons, Raytheon, and Siemens.

Companies and projects with excellent quality control tend to have low levels of code cyclomatic complexity and high test coverage; i.e. test cases cover > 95% of paths and risk areas.

These companies also measure quality well and all know their defect removal efficiency (DRE) levels. (Any company that does not measure and know their DRE is probably below 85% in DRE.)

Excellent quality control has defect removal efficiency levels (DRE) between about 97% for large systems in the 10,000 function point size range and about 99.6% for small projects < 1,000 function points in size.

A DRE of 100% is theoretically possible but is extremely rare. The author has only noted DRE of 100% in two projects out of a total of about 20,000 projects examined.

Average Quality Control

In today's world agile is the new average. Agile development has proven to be effective for smaller applications below 1,000 function points in size. Agile does not scale up well and is not a top method for quality. Agile is weak in quality measurements and does not normally use inspections, which has the highest defect removal efficiency (DRE) of any known form of defect removal. Inspections top 85% in DRE and also raise testing DRE levels. Among the authors clients that use Agile the average value for defect removal efficiency is about 92%. This is certainly better than the 85% industry average, but not up to the 99% actually needed to achieve optimal results.

Some but not all agile projects use "pair programming" in which two programmers share an office and a work station and take turns coding while the other watches and "navigates." Pair programming is very expensive but only benefits quality by about 15% compared to single programmers. Pair programming is much less effective in finding bugs than formal inspections, which usually bring 3 to 5 personnel together to seek out bugs using formal methods.

Agile is a definite improvement for quality compared to waterfall development, but is not as effective as the quality-strong methods of team software process (TSP) and the rational unified process (RUP).

Average projects usually do not know defects by origin, and do not measure defect removal efficiency until testing starts; i.e. requirements and design defects are under reported and sometimes invisible.

A recent advance in software quality control now frequently used by average as well as advanced organizations is that of static analysis. Static analysis tools can find about 55% of code defects, which is much higher than most forms of testing.

Many test stages such as unit test, function test, regression test, etc. are only about 35% efficient in finding code bugs, or find one bug out of three. This explains why 6 to 10 separate kinds of testing are needed.

The kinds of companies and projects that are "average" would include internal software built by hundreds of banks, insurance companies, retail and wholesale companies, and many government agencies at federal, state, and municipal levels.

Average quality control has defect removal efficiency levels (DRE) from about 85% for large systems up to 97% for small and simple projects.

Poor Quality Control

Poor quality control is characterized by weak defect prevention and almost a total omission of pre-test defect removal methods such as static analysis and formal inspections. Poor quality control is also characterized by inept and inaccurate quality measures which ignore front-end defects in requirements and design. There are also gaps in measuring code defects. For example most companies with poor quality control have no idea how many test cases might be needed or how efficient various kinds of test stages are.

Companies with poor quality control also fail to perform any kind of up-front quality predictions so they jump into development without a clue as to how many bugs are likely to occur and what are the best methods for preventing or removing these bugs.

One of the main reasons for the long schedules and high costs associated with poor quality is the fact that so many bugs are found when testing starts that the test interval stretches out to two or three times longer than planned.

Some of the kinds of software that are noted for poor quality control include the Obamacare web site, municipal software for property tax assessments, and software for programmed stock trading, which has caused several massive stock crashes.

Poor quality control is below 85% in defect removal efficiency (DRE) levels. In fact for canceled projects or those that end up in litigation for poor quality, the DRE levels may drop below 80%, which is low enough to be considered professional malpractice. In litigation where the author has been an expert witness DRE levels in the low 80% range have been the unfortunate norm.

Reuse of Certified Materials for Software Projects

So long as software applications are custom designed and coded by hand, software will remain a labor-intensive craft rather than a modern professional activity. Manual software development even with excellent methodologies cannot be much more than 15% better than average development due to the intrinsic limits in human performance and legal limits in the number of hours that can be worked without fatigue.

The best long-term strategy for achieving consistent excellence at high speed would be to eliminate manual design and coding in favor of construction from certified reusable components.

It is important to realize that software reuse encompasses many deliverables and not just source code. A full suite of reusable software components would include at least the following 10 items:

1. **Reusable requirements**
2. **Reusable architecture**
3. **Reusable design**
4. **Reusable code**
5. **Reusable project plans and estimates**
6. **Reusable test plans**
7. **Reusable test scripts**
8. **Reusable test cases**
9. **Reusable user manuals**
10. **Reusable training materials**

These materials need to be certified to near zero-defect levels of quality before reuse becomes safe and economically viable. Reusing buggy materials is harmful and expensive. This is why excellent quality control is the first stage in a successful reuse program.

The need for being close to zero defects and formal certification adds about 20% to the costs of constructing reusable artifacts, and about 30% to the schedules for construction. However using certified reusable materials subtracts over 80% from the costs of construction and can shorten schedules by more than 60%. The more times materials are reused the greater their cumulative economic value.

One caution to readers: reusable artifacts may be treated as taxable assets by the Internal Revenue Service. It is important to check this topic out with a tax attorney to be sure that formal corporate reuse programs will not encounter unpleasant tax consequences.

The three samples in table 1 showed only moderate reuse typical for the end of 2013: Excellent project (15% certified reuse - close to current maximum); Average project (10% certified reuse); and Poor projects (5% certified reuse).

In the future it is technically possible to make large increases in the volumes of reusable materials. By around 2025 we should be able to construct software applications with perhaps 85% certified reusable materials.

Table 2 shows the productivity impact of increasing volumes of certified reusable materials. Table 2 uses whole numbers and generic values to simplify the calculations.

Software reuse from certified components instead of custom design and hand coding is the only known technique that can achieve order-of-magnitude improvements in software productivity. True excellence in software engineering must derive from replacing costly and error-prone manual work with construction from certified reusable components.

Because finding and fixing bugs is the major software cost driver, increasing volumes of high-quality certified materials can convert software from an error-prone manual craft into a very professional high-technology profession. Table 3 shows probable quality gains from increasing volumes of software reuse.

Reuse Percent	Months of staff effort	Function Points per month	Work hours per function point	Lines of Code per month	Project Costs
0.00%	100	10.00	13.20	533	$1,000,000
10.00%	90	11.11	11.88	592	$900,000
20.00%	80	12.50	10.56	666	$800,000
30.00%	70	14.29	9.24	761	$700,000
40.00%	60	16.67	7.92	888	$600,000
50.00%	50	20.00	6.60	1,066	$500,000
60.00%	40	25.00	5.28	1,333	$400,000
70.00%	30	33.33	3.96	1,777	$300,000
80.00%	20	50.00	2.64	2,665	$200,000
90.00%	10	100.00	1.32	5,330	$100,000
100.00%	1	1,000.00	0.13	53,300	$10,000

Table 2: Productivity Gains from Software Reuse (Assumes 1000 function points and 53,300 LOC)

Reuse Percent	Defects per Function Point	Defect Potential	Defect Removal Efficiency	Delivered Defects
0.00%	5.00	1,000	90.00%	100
10.00%	4.50	900	91.00%	81
20.00%	4.00	800	92.00%	64
30.00%	3.50	700	93.00%	49
40.00%	3.00	600	94.00%	36
50.00%	2.50	500	95.00%	25
60.00%	2.00	400	96.00%	16
70.00%	1.50	300	97.00%	9
80.00%	1.00	200	98.00%	4
90.00%	0.50	100	99.00%	1
100.00%	-	1	99.99%	0

Table 3: Quality Gains from Software Reuse (Assumes 1,000 function points and 53,300 LOC)

Since the current maximum for software reuse from certified components is only in the range of 15% or a bit higher, it can be seen that there is a large potential for future improvement.

Note that uncertified reuse in the form of mashups or extracting materials from legacy applications may top 50%. However uncertified reusable materials often have latent bugs, security flaws, and even error-prone modules so this not a very safe practices. In several cases the reused material was so buggy it had to be discarded and replaced by custom development.

Software Methodologies

Unfortunately selecting a methodology is more like joining a cult than making an informed technical decision. Most companies don't actually perform any kind of due diligence on methodologies and merely select the one that is most popular.

In today's world agile is definitely the most popular. Fortunately agile is also a pretty good methodology and much superior to the older waterfall method. However there are some caveats about methodologies.

Agile has been successful primarily for smaller applications < 1,000 function points in size. It has also been successful for internal applications where users can participate or be "embedded" with the development team to work our requirements issues.

Agile has not scaled up well to large systems > 10,000 function points. Agile has also not been visibly successful for commercial or embedded applications where there are millions of users and none of them work for the company building the software so their requirements have to be collected using focus groups or special marketing studies.

A variant of agile that uses "pair programming" or two programmers working in the same cubical with one coding and the other "navigating" has become popular. However it is very expensive since two people are being paid to do the work of one person. There are claims that quality is improved, but formal inspections combined with static analysis achieve much higher quality for much lower costs.

Another agile variation, extreme programming, in which test cases are created before the code itself is written has proven to be fairly successful for both quality and productivity, compared to traditional waterfall methods. However both TSP and RUP are just as good and even better for large systems.

There are dozens of available methodologies circa 2013 and many are good; some are better than agile for large systems; some older methods such as waterfall and cowboy development are at the bottom of the effectiveness list and should be avoided on modern applications.

For major applications in the 10,000 function point size range and above the team software process (TSP) and the Rational unified process (RUP) have the best track records for successful projects and among the fewest failures.

Quantifying Software Excellence

Because the software industry has a poor track record for measurement, it is useful to show what "excellence" means in quantified terms.

Excellence in software quality combines defect potentials of no more than 2.5 bugs per function point combined with defect removal efficiency (DRE) of 99%. This means that delivered defects will not exceed 0.025 defects per function point.

By contrast current average values circa 2013 are about 3.0 to 5.0 bugs per function point for defect potentials and only 85% to 90% DRE, leading to as many as 0.75 bugs per function point at delivery.

Excellence in software productivity and schedules is not a fixed value but varies with the size of the applications. Table 4 shows two "flavors" of productivity excellence: 1) the best that can be accomplished with 10% reuse and 2) the best that can be accomplished with 50% reuse:

As can be seen from table 4, software reuse is the most important technology for improving software productivity and quality by really significant amounts. Methods, tools, CMMI levels, and other minor factors are certainly beneficial. However so long as software applications are custom designed and hand coded software will remain an expensive craft and not a true professional occupation.

Summary and Conclusions

Because software is the driving force of both industry and government operations, it needs to be improved in terms of both quality and productivity. The most powerful technology for making really large improvements in both quality and productivity will be from eliminating costly custom designs and labor-intensive hand coding, and moving towards manufacturing software applications from libraries of well-formed standard reusable components that approach zero-defect quality levels.

Today's best combinations of methods, tools, and programming languages are certainly superior to waterfall or cowboy development using unstructured methods and low-level languages. But even the best current methods still involve error-prone custom designs and labor-intensive manual coding.

	Schedule Months	Staffing	Effort Months	FP per Month
With < 10% certified reuse				
100 function points	4.79	1.25	5.98	16.71
1,000 function points	13.80	6.25	86.27	11.59
10,000 function points	33.11	57.14	1,892.18	5.28
100,000 function points	70.79	540.54	38,267.34	2.61
With 50% certified reuse				
100 function points	3.98	1.00	3.98	25.12
1,000 function points	8.51	5.88	50.07	19.97
10,000 function points	20.89	51.28	1,071.43	9.33
100,000 function points	44.67	487.80	21,789.44	4.59

Table 4: Excellent Productivity with Varying Quantities of Certified Reuse

Disclaimers:

ABOUT THE AUTHOR

Capers Jones is currently vice president and chief technology officer of Namcook Analytics LLC. The company designs leading-edge risk, cost, and quality estimation and measurement tools.

Prior to the formation of Namcook Analytics in 2012 Capers Jones was the president of Capers Jones & Associates LLC between 2000 and 2012.

He is also the founder and former chairman of Software Productivity Research LLC (SPR). Capers Jones founded SPR in 1984 and sold the company to Artemis Management Systems in 1998. He was the chief scientist at Artemis until retiring in 2000. SPR marketed three successful commercial estimation tools: SPQR/20 in 1984; CheckPoint in 1995; and KnowledgePlan in 1998. SPR also built custom proprietary estimation tools for AT&T and Bachman Systems.

Before founding SPR Capers was Assistant Director of Programming Technology for the ITT Corporation at the Programming Technology Center in Stratford, Connecticut. During his tenure Capers Jones designed three proprietary software cost and quality estimation tools for ITT between 1979 and 1983.

He was also a manager and software researcher at IBM in California where he designed IBM's first two software cost estimating tools in 1973 and 1974 in collaboration with Dr. Charles Turk.

Capers Jones is a well-known author and international public speaker. Some of his books have been translated into five languages. His five most recent books are "The Technical and Social History of Software Engineering," Addison Wesley 2014; "The Economics of Software Quality with Olivier Bonsignour," Addison Wesley, 2011; "Software Engineering Best Practices," McGraw Hill 2010; "Applied Software Measurement," 3rd edition, McGraw Hill, 2008; and "Estimating Software Costs," McGraw Hill, 2nd edition, 2007.

Among his older book titles are "Patterns of Software Systems Failure and Success" (Prentice Hall 1994); Estimating Software Risks," International Thomson 1995; "Software Quality: Analysis and Guidelines for Success" (International Thomson 1997); and "Software Assessments: Benchmarks, and Best Practices" (Addison Wesley Longman 2000).

Capers and his colleagues have collected historical data from thousands of projects, hundreds of corporations, and more than 30 government organizations. This historical data is a key resource for judging the effectiveness of software process improvement methods and also for calibrating software estimation accuracy.

Capers Jones data is also widely cited in software litigation in cases where quality, productivity, and schedules are part of the proceedings. Capers Jones has also worked as an expert witness in 15 lawsuits involving breach of contract and software taxation issues.

Phone: 401-864-2632
E-mail: Capers.Jones3@gmail.com
Blog: http://Namcookanalytics.com
Web: www.Namcook.com

REFERENCES

1. Abran, A. and Robillard, P.N.; "Function Point Analysis, An Empirical Study of its Measurement Processes"; IEEE Transactions on Software Engineering, Vol 22, No. 12; Dec. 1996; pp. 895-909.
2. Austin, Robert d:; Measuring and Managing Performance in Organizations; Dorset House Press, New York, NY; 1996; ISBN 0-932633-36-6; 216 pages.
3. Black, Rex; Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing; Wiley; 2009; ISBN-10 0470404159; 672 pages.
4. Bogan, Christopher E. and English, Michael J.; Benchmarking for Best Practices; McGraw Hill, New York, NY; ISBN 0-07-006375-3; 1994; 312 pages.
5. Brown, Norm (Editor); The Program Manager's Guide to Software Acquisition Best Practices; Version 1.0; July 1995; U.S. Department of Defense, Washington, DC; 142 pages.
6. Cohen, Lou; Quality Function Deployment – How to Make QFD Work for You; Prentice Hall, Upper Saddle River, NJ; 1995; ISBN 10: 0201633302; 368 pages.
7. Crosby, Philip B.; Quality is Free; New American Library, Mentor Books, New York, NY; 1979; 270 pages.
8. Curtis, Bill, Hefley, William E., and Miller, Sally; People Capability Maturity Model; Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA; 1995.
9. Department of the Air Force; Guidelines for Successful Acquisition and Management of Software Intensive Systems; Volumes 1 and 2; Software Technology Support Center, Hill Air Force Base, UT; 1994.
10. Dreger, Brian; Function Point Analysis; Prentice Hall, Englewood Cliffs, NJ; 1989; ISBN 0-13-332321-8; 185 pages.
11. Gack, Gary; Managing the Black Hole: The Executives Guide to Software Project Risk; Business Expert Publishing, Thomson, GA; 2010; ISBN10: 1-935602-01-9.
12. Gack, Gary; Applying Six Sigma to Software Implementation Projects; <http://software.isixsigma.com/library/content/c040915b.asp.>
13. Gilb, Tom and Graham, Dorothy; Software Inspections; Addison Wesley, Reading, MA; 1993; ISBN 10: 0201631814.
14. Grady, Robert B.; Practical Software Metrics for Project Management and Process Improvement; Prentice Hall, Englewood Cliffs, NJ; ISBN 0-13-720384-5; 1992; 270 pages.
15. Grady, Robert B. & Caswell, Deborah L.; Software Metrics: Establishing a Company-Wide Program; Prentice Hall, Englewood Cliffs, NJ; ISBN 0-13-821844-7; 1987; 288 pages.
16. Grady, Robert B.; Successful Process Improvement; Prentice Hall PTR, Upper Saddle River, NJ; ISBN 0-13-626623-1; 1997; 314 pages.
17. Humphrey, Watts S.; Managing the Software Process; Addison Wesley Longman, Reading, MA; 1989.
18. IFPUG Counting Practices Manual, Release 4, International Function Point Users Group, Westerville, OH; April 1995; 83 pages.
19. Jacobsen, Ivar, Griss, Martin, and Jonsson, Patrick; Software Reuse - Architecture, Process, and Organization for Business Success; Addison Wesley Longman, Reading, MA; ISBN 0-201-92476-5; 1997; 500 pages.
20. Jones, Capers and Bonsignour, Olivier; The Economics of Software Quality, Addison Wesley Longman, Reading, MA; 2011.
21. Jones, Capers; Estimating Software Costs; 2nd edition; McGraw Hill; New York, NY; 2007.
22. Jones, Capers; Software Engineering Best Practices; McGraw Hill, New York, NY; 2010.
23. Jones, Capers and Bonsignour, Olivier; The Economics of Software Quality; Addison Wesley, Boston, MA; 2011; ISBN 978-0-13-258220-9; 587 pages.
24. Jones, Capers; "A Ten-Year Retrospective of the ITT Programming Technology Center"; Software Productivity Research, Burlington, MA; 1988.
25. Jones, Capers; Applied Software Measurement; McGraw Hill, 3rd edition 2008.
26. Jones, Capers; Software Engineering Best Practices; McGraw Hill, 1st edition 2010.
27. Jones, Capers; Assessment and Control of Software Risks; Prentice Hall, 1994; ISBN 0-13-741406-4; 711 pages.
28. Jones, Capers; Patterns of Software System Failure and Success; International Thomson Computer Press, Boston, MA; December 1995; 250 pages; ISBN 1-850 -32804-8; 292 pages.

29. Jones, Capers; Software Assessments, Benchmarks, and Best Practices; Addison Wesley Longman, Boston, MA; 2000 (due in May of 2000); 600 pages.
30. Jones, Capers; Software Quality – Analysis and Guidelines for Success; International Thomson Computer Press, Boston, MA; ISBN 1-85032-876-6; 1997; 492 pages.
31. Jones, Capers; The Economics of Object-Oriented Software; Software Productivity Research, Burlington, MA; April 1997; 22 pages.
32. Jones, Capers; Becoming Best in Class; Software Productivity Research, Burlington, MA; January 1998; 40 pages.
33. Kan, Stephen H.; Metrics and Models in Software Quality Engineering; 2nd edition; Addison Wesley Longman, Boston, MA; ISBN 0-201-72915-6; 2003; 528 pages.
34. Keys, Jessica; Software Engineering Productivity Handbook; McGraw Hill, New York, NY; ISBN 0-07-911366-4; 1993; 651 pages.
35. Love, Tom; Object Lessons; SIGS Books, New York; ISBN 0-9627477 3-4; 1993; 266 pages.
36. McCabe, Thomas J.; "A Complexity Measure"; IEEE Transactions on Software Engineering; December 1976; pp. 308-320.
37. Melton, Austin; Software Measurement; International Thomson Press, London, UK; ISBN 1-85032-7178-7; 1995.
38. Multiple authors; Rethinking the Software Process; (CD-ROM); Miller Freeman, Lawrence, KS; 1996. (This is a new CD ROM book collection jointly produced by the book publisher, Prentice Hall, and the journal publisher, Miller Freeman. This CD ROM disk contains the full text and illustrations of five Prentice Hall books: Assessment and Control of Software Risks by Capers Jones; Controlling Software Projects by Tom DeMarco; Function Point Analysis by Brian Dreger; Measures for Excellence by Larry Putnam and Ware Myers; and Object-Oriented Software Metrics by Mark Lorenz and Jeff Kidd.)
39. Paulk Mark et al; The Capability Maturity Model; Guidelines for Improving the Software Process; Addison Wesley, Reading, MA; ISBN 0-201-54664-7; 1995; 439 pages.
40. Perry, William E.; Data Processing Budgets - How to Develop and Use Budgets Effectively; Prentice Hall, Englewood Cliffs, NJ; ISBN 0-13-196874-2; 1985; 224 pages.
41. Perry, William E.; Handbook of Diagnosing and Solving Computer Problems; TAB Books, Inc.; Blue Ridge Summit, PA; 1989; ISBN 0-8306-9233-9; 255 pages.
42. Putnam, Lawrence H.; Measures for Excellence -- Reliable Software On Time, Within Budget; Yourdon Press - Prentice Hall, Englewood Cliffs, NJ; ISBN 0-13-567694-0; 1992; 336 pages.
43. Putnam, Lawrence H and Myers, Ware.; Industrial Strength Software - Effective Management Using Measurement; IEEE Press, Los Alamitos, CA; ISBN 0-8186-7532-2; 1997; 320 pages.
44. Radice, Ronald A.; High Quality Low Cost Software Inspections; Paradoxicon Publishingl Andover, MA; ISBN 0-9645913-1-6; 2002; 479 pages.
45. Royce, Walker E.; Software Project Management: A Unified Framework; Addison Wesley Longman, Reading, MA; 1998; ISBN 0-201-30958-0.
46. Rubin, Howard; Software Benchmark Studies For 1997; Howard Rubin Associates, Pound Ridge, NY; 1997.
47. Rubin, Howard (Editor); The Software Personnel Shortage; Rubin Systems, Inc.; Pound Ridge, NY; 1998.
48. Shepperd, M.: "A Critique of Cyclomatic Complexity as a Software Metric"; Software Engineering Journal, Vol. 3, 1988; pp. 30-36.
49. Strassmann, Paul; The Squandered Computer; The Information Economics Press, New Canaan, CT; ISBN 0-9620413-1-9; 1997; 426 pages.
50. Stukes, Sherry, Deshoretz, Jason, Apgar, Henry and Macias, Ilona; Air Force Cost Analysis Agency Software Estimating Model Analysis ; TR-9545/008-2; Contract F04701-95-D-0003, Task 008; Management Consulting & Research, Inc.; Thousand Oaks, CA 91362; September 30 1996.
51. Symons, Charles R.; Software Sizing and Estimating – Mk II FPA (Function Point Analysis); John Wiley & Sons, Chichester; ISBN 0 471-92985-9; 1991; 200 pages.
52. Thayer, Richard H. (editor); Software Engineering and Project Management; IEEE Press, Los Alamitos, CA; ISBN 0 8186-075107; 1988; 512 pages.
53. Umbaugh, Robert E. (Editor); Handbook of IS Management; (Fourth Edition); Auerbach Publications, Boston, MA; ISBN 0-7913-2159-2; 1995; 703 pages.
54. Weinberg, Dr. Gerald; Quality Software Management - Volume 2 First-Order Measurement; Dorset House Press, New York, NY; ISBN 0-932633-24-2; 1993; 360 pages.
55. Wiegers, Karl A; Creating a Software Engineering Culture; Dorset House Press, New York, NY; 1996; ISBN 0-932633-33-1; 358 pages.
56. Yourdon, Ed; Death March - The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects; Prentice Hall PTR, Upper Saddle River, NJ; ISBN 0-13-748310-4; 1997; 218 pages.
57. Zells, Lois; Managing Software Projects - Selecting and Using PC-Based Project Management Systems; QED Information Sciences, Wellesley, MA; ISBN 0-89435-275-X; 1990; 487 pages.
58. Zvegintzov, Nicholas; Software Management Technology Reference Guide; Dorset House Press, New York, NY; 1994; ISBN 1-884521-01-0; 240 pages.

CALL FOR ARTICLES

If your experience or research has produced information that could be useful to others, CrossTalk can get the word out. We are specifically looking for articles on software-related topics to supplement upcoming theme issues. Below is the submittal schedule for two areas of emphasis we are looking for:

Software Education Today
Jan/Feb 2015 Issue
Submission Deadline: Aug 10, 2014

Test and Diagnostics
Mar/Apr 2015 Issue
Submission Deadline: Oct 10, 2014

Please follow the Author Guidelines for CrossTalk, available on the Internet at <www.crosstalkonline.org/submission-guidelines>. We accept article submissions on software-related topics at any time, along with Letters to the Editor and BackTalk. To see a list of themes for upcoming issues or to learn more about the types of articles we're looking for visit <www.crosstalkonline.org/theme-calendar>.

Improving Software through Metrics while Providing Cradle to Grave Support

Jennifer Walters, Northrop Grumman
Kevin MacG. Adams, Ph.D., NCSOSE

Abstract. Metrics are beneficial to an organization that supports a product from inception through product retirement and disposal. Quality metrics have a critical role in this type of environment because they span both the development and operations and maintenance phases of the software life cycle, and there is a relationship between the internal quality metrics collected during development and the external quality metrics collected once the product is deployed. The key finding is that internal metrics can be collected early in the software development phase to predict the support required during the operations and maintenance phase; likewise, external metrics can be collected to drive software development process improvements. Finally, analyzing the relationships between the two can drive overall process improvements for the entire software lifecycle.

Introduction

Software development is "the specification, construction, testing and delivery of a new application or of a discrete addition to an existing application" [1] while a maintenance project is "a software development project described as maintenance to correct errors in an original requirements specification, to adapt a system to a new environment, or to enhance a system" [1]. Often times these two processes are supported by two different organizations with two different goals. One team will focus on building the initial application and their primary concern is building a product that fulfills the requirements within both cost and schedule constraints. The second team has the responsibility of supporting the product during the remainder of the lifetime and has a primary concern of maintainability.

As a result, the maintenance team is at the mercy of the software design and processes imposed by the software development team. While the resulting product might meet all the user requirements and appear to be a superb product, it is likely that the software development team did not build the initial product with maintenance in mind [2]. This results in a product that is more difficult and costly to maintain.

One Organization Supporting Entire Lifecycle

When the same organization supports both the development phase and the operations and maintenance phase, however, there are some opportunities to create a synergy between development and maintenance efforts. The focus can shift from individual phases to the overall software lifecycle. By transitioning focus, team members can collect measurements early in the development phase that can help predict issues in maintenance. Likewise, maintenance related metrics can be analyzed to provide guidance for improving development processes. Cause and effect analysis is a very powerful technique in this situation.

Role of Metrics

It is important to understand the role metrics play in the overall software lifecycle. First, the metrics of concern in this paper are quality attributes because maintenance efforts are highly dependent on the overall quality of the software [3]. In addition, quality attributes span both pre-delivery and post-delivery phases of the lifecycle and are therefore specifically relevant when a single organization supports the software over the entire lifecycle. Second, quality attributes are categorized as either internal or external [4]. "Internal quality attributes are those that can be directly measured purely on the basis of product features such as size, length, or complexity" [5]. External metrics are measurements that are dependent on how the software interacts with its environment and can therefore only be collected after the product has been deployed and operated during the maintenance and operations phase of the software lifecycle [5]. The remainder of this discussion involves the relationship between internal and external metrics.

Internal Metrics

As previously mentioned, internal quality attributes are measurements based on characteristics of the product itself. Size, length, and complexity are the key attributes collected. Research has shown that there is a correlation between internal quality attributes and external quality of the product [5-8]. By measuring internal quality that can be accomplished early during the software development process, it is possible to predict product maintainability and thus the effort required to support product maintenance.

There are many metric options available for evaluating internal software design quality. A few of the most commonly used suites include: Chidamber and Kemerer (CK) Metrics [9], Robert C. Martin Metric Suite [10], and McCabe's Metric Suite [11,12]. Table 1 provides a brief overview of the metric suites. The next section will discuss CK Metrics in more detail.

Chidamber and Kemerer Metrics

Chidamber and Kemerer (CK) metrics can "assist users in understanding object oriented design complexity and in forecasting external software qualities for example software defects, testing, and maintenance effort" [13]. Numerous research studies have validated CK metrics as a method for predicting maintainability [8, 13-16]. The suite includes six metrics: (1) weighted methods per class (WMC), (2) depth of inheritance tree (DIT), (3) number of children (NOC), (4) coupling between objects/classes (CBO), (5) response for a class (RFC), and (6) lack of cohesion in method (LCOM). Each of the six measurements in the CK suite quantify different quality attributes which relate to maintainability qualities; however, the last metric discussed in the next section, LCOM, has been shown to have the greatest impact on the total number of defects [8].

Weighted Methods per Class

Weighted Methods per Class (WMC) is a complexity measurement. However, Chidamber and Kemerer did not propose

a definition or method for measuring complexity. "If methods complexities are considered to be unity, the WMC metric turns in to the number of methods in a class" [17]. Whether it is generalized to a count of methods or is more specialized through the use of a complexity algorithm, a higher WMC indicates a class that is more difficult to understand and modify.

Depth of Inheritance Tree

Depth of Inheritance Tree (DIT) is object oriented (OO) specific as it measures the OO characteristic inheritance. Inheritance is "a semantic notion by which the responsibilities (properties and constraints) of a subclass are considered to include the responsibilities of a superclass, in addition to its own, specifically declared responsibilities" [1]. It is often described as an isa relationship. For example, a cat isa mammal. This can be extended to include a mammal isa animal. DIT measures the hierarchy of inheritance. In this example, there is a hierarchy of three: cat > mammal > animal. The deeper a class is in the inheritance tree, the more difficult it is to comprehend and predict the behavior of the class [18].

Number of Children

Number of Children (NOC) is similar to DIT as it is related to inheritance. It is the count of immediate sub-classes in the class hierarchy [13]. NOC takes a more horizontal approach to inheritance. Instead of walking down the inheritance tree, it counts the number of classes inheriting methods from the parent class. Extending the previous example, a reptile is also an animal. Now both mammal and reptile classes inherit the members from animal. So, if the animal base class is modified there is potential impact to both sub-classes. High NOC measurements require more impact analysis.

Coupling Between Objects Classes

As the name suggests, Coupling Between Objects Classes (CBO) measures coupling. Coupling is "the manner and degree of interdependence between software modules" [1]. A class is considered coupled with another class if its members are used by another class. Coupling makes it more difficult to isolate units of code for testing. The interdependence also makes code comprehension more difficult and increases the need for impact analysis. It is "highly connected to portability, maintainability, and re-usability" [17].

Response for a Class

Response for a Class (RFC) is similar to CBO but this measure also takes inheritance into count. It is the number of methods that can be executed as a response to a message received by class objects [13,17]. The count includes methods in the same class, methods accessible within the class hierarchy, and methods accessible in other classes. Source code with a high RFC count is complex and can be very difficult to trace potential code paths for testing and comprehension.

Metric Suite	Description	Measurements
CK Metrics	CK metrics designed specifically for object-oriented software [9]. Commonly utilized to predict fault-proneness [8] and included in static code analysis tools to provide automation opportunities.	• Weighted Methods Per Class (WMC) • Depth of Inheritance Tree (DIT) • Number of Immediate Subclasses (NOC) • Coupling between Objects Classes (CBO) • Response for a Class (RFC) • Lack of Cohesion in Method (LCOM)
Robert C. Martin Metric Suite	This suite of metrics focuses on interdependence between packages, or cohesion [10]. It is also designed for object-oriented software.	• Afferent Coupling • Efferent Coupling • Instability • Abstractness • Normalized Distance from Main Sequence
McCabe's Metric Suite	McCabe is most commonly associated with the concept of cyclomatic complexity. The metric was introduced as a way to quantify design decisions to indicate how difficult it is to test and maintain the method [11, 12]. Also commonly found in static analysis tools.	• McCabe's Complexity • Method Lines of Code • Total Lines of Code • Nested Block Depth

Table 1: Metric Suites for Measuring Internal Quality Attributes

Lack of Cohesion in Method

The final metric in the CK suite is Lack of Cohesion in Method (LCOM). Since cohesion is considered a positive characteristic in object-oriented code, this measurement considers the lack of cohesion. "A highly cohesive module should be independent" [13] of other modules and improve reusability. Lack of cohesion on the other hand signifies poor design that generates more complex code that is difficult to maintain. It is an indicator of potential redesign opportunities to create smaller more cohesive classes [13,17]. LCOM "has a significant effect on the total number of defects… and software development companies should concentrate on [it] to control… design defects" [8].

External Metrics

While internal metrics provide a wealth of interesting information, the measures themselves are not very helpful on their own. "For these early indicators to be meaningful, they must be related (in a statistically significant and stable way) to the field quality/reliability of the product" [7]. The field quality attributes can only be measured while the product is in the testing or operations and maintenance phase of the software lifecycle (ISO/IEC, 2002) and are known as external metrics. Figure 1 displays the relation-

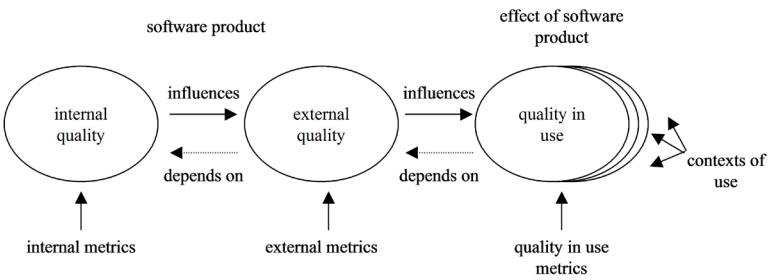


Figure 1: Relationships between types of metrics [19, p.4]

While much emphasis is placed on the development phase, most software products do eventually reach the operations and maintenance phase where they must then be maintained until the system is no longer needed or is replaced by a new product. It stands to reason, then, that software should be developed with maintenance in mind. But this is not typically the case because the software development team is focused on quickly creating a product to meet the needs of the customer at a cost and schedule the customer is willing to accept. The team is likely not considering how the product will be maintained after it is in operation.

ship between the internal metrics already discussed and external metrics which include functionality, usability, reliability, maintainability, portability, and efficiency [19].

As shown, internal metrics influence external quality and external quality depends on internal metrics. With this type of relationship it is possible to predict external quality as well as maintenance support needs of the final product by collecting internal metrics which are available earlier in the software lifecycle [7,8,13,19].

Maintainability

While all of the external metrics are important to a maintenance organization, maintainability is critical when planning and staffing maintenance activities and will be the focus of the external metric discussion that follows. Maintainability is "the ease with which software system or component can be modified to change or add capabilities, correct faults or defects, improve performance or other attributes, or adapt to a changed environment" [1]. The four primary maintainability metrics are analyzability, changeability, stability, and testability.

Analyzability

Analyzability describes the ability to trace and comprehend the intent of the existing software code. The maintenance organization must be able to comprehend the existing software in order to perform analysis activities [20] such as defect analysis and impact analysis. All six of the CK Metrics discussed earlier have a great impact on analyzability. From lack of cohesion to highly coupled modules to deep inheritance trees, all of these factors contribute to the difficulty of reading, tracing, and comprehending the code base. Some options for measuring analyzability are audit trail capability, diagnostic function support, failure analysis capability, failure analysis efficiency, and status monitoring capability [19].

Changeability

Changeability metrics relate to the actual maintenance activities that modify the existing code. The modification could be the result of discovering a defect, the need to adapt to a new environment, or the request for a new enhancement. Regardless of the reason for the change, most of the same qualities that make a code easier to analyze also makes it easier to change. High cohesion and low coupling are critical when developing an application with a focus on maintenance. An organization should review the relationship between internal metrics and changeability by gathering some of the following external metrics: change cycle efficiency, change implementation elapse time, modification complexity, parameterized modifiability, and software change control capability [19].

Stability

Most users dislike unexpected software behavior. Stability related metrics such as change success ratio and modification impact localization [19] provide evidence of unexpected behavior or the lack thereof. One method for decreasing the risk of unexpected behavior is to complete a thorough impact analysis and ensure modified code paths are systematically tested. With this understanding it is clear that the internal metrics that impact analyzability and testability are also crucial to predicting software stability.

Testability

The final maintainability component discussed is testability. As its name suggests, testability refers to the ease at which the software can be tested, or verified and validated. To measure testability, an organization should collect the following measurements: availability of built-in test function, re-test efficiency, and test restartability [19]. Coupling is a critical characteristic when determining testability [21]. When modules are coupled, it is more difficult or impossible to isolate the module under test to ensure the test is focusing only on the desired code and producing clear concise results. CBO and RFC metrics will have a strong correlation to the external metrics related to testability.

Conclusion

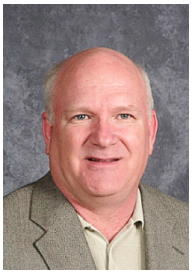
Software operations and maintenance is a critical phase in the software lifecycle. While much emphasis is placed on the development phase, most software products do eventually reach the operations and maintenance phase where they must then be maintained until the system is no longer needed or is replaced by a new product. It stands to reason, then, that software should be developed with maintenance in mind. But this is not typically the case because the software development team is focused on quickly creating a product to meet the needs of the customer at a cost and schedule the customer is willing to accept. The team is likely not considering how the product will be maintained after it is in operation.

However, when the same organization supports both software development and software operations and maintenance, there is incentive to keep maintenance in mind during the development process. It is also easier to capture both internal metrics, such as CK Metrics, that are available during the development phases as well as external metrics which are available only after the product is in operation. Internal metrics can help predict the level of support required to maintain the product once it is in operation. Furthermore, by analyzing the relationships between these two sets of metrics, the organization can improve both development and maintenance processes and thus improves the overall quality of the product as it supports the software from the cradle to the grave.

ABOUT THE AUTHORS

Jennifer Walters is a software development analyst at Northrop Grumman Enterprise Shared Services. She holds a B.S. in Computer and Information Science from University of Maryland University College, an M.A.E.D. in Secondary Education from University of Phoenix, and an M.S. in Information Technology from University of Maryland University College.

RR 2 BX 542
Nelson Drive
Ridgeley, WV 26753
E-mail: jennifer.walters@ngc.com
Phone: 304-726-4174 (home)



Dr. Kevin MacG. Adams is an Adjunct Professor at the University of Maryland University College where he teaches software and systems engineering in the graduate program in Information Technology. Dr. Adams is a retired Navy submarine officer and information systems consultant. Dr. Adams holds a B.S. in Ceramic Engineering from Rutgers University, an M.S. in Naval Architecture and Marine Engineering and an M.S. in Materials Engineering both from MIT, and a Ph.D. in Systems Engineering from Old Dominion University.

University of Maryland University College
3501 University Blvd. East,
Adelphi, Maryland 20783
E-mail: kevin.adams@faculty.umuc.edu
Phone: 757-855-1954 (home)

REFERENCES

1. IEEE, IEEE and ISO/IEC Standard 24765: Systems and software engineering – Vocabulary. New York and Geneva: Institute of Electrical and Electronics Engineers and the International Organization for Standardization and the International Electrotechnical Commission, 2010.
2. A. Abran and J. Moore, Guide to the Software Engineering Body of Knowledge, 2004 ed. Los Alamitos, CA: The Institute of Electrical and Electronics Engineers, 2004.
3. J. Bansiya and C. Davis, "A hierarchical model for object-oriented design quality assessment," IEEE Transactions on Software Engineering, vol. 28, pp. 4-17, 2002.
4. M. Barbacci, M. Klein, T. Longstaff, and C. Weinstock, "Quality Attributes (Technical Report: CMU/SEI-95-TR-021, ESC-TR-95-021)," Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA1995.
5. E. Bagheri and D. Gasevic, "Assessing the maintainability of software product line feature models using structural metrics," Software Quality Journal, vol. 19, pp. 579-612, 2011.
6. M. Bocco, D. L. Moody, and M. Piattini, "Assessing the capability of internal metrics as early indicators of maintenance effort through experimentation," Journal of Software Maintenance & Evolution: Research & Practice, vol. 17, pp. 225-246, 2005.
7. N. Nagappan, "Toward a software testing and reliability early warning metric suite," in Proceedings of the 26th International Conference on Software Engineering, ed Los Alamitos, CA: IEEE Computer Society, 2004, pp. 60-62.
8. M. R. J. Qureshi and W. Qureshi, "Evaluation of the design metric to reduce the number of defects in software development," International Journal of Information Technology and Computer Science, vol. 4, pp. 9-17, 2012.
9. S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in Proceedings of the Conference on Object-oriented programming systems, languages, and applications (OOPSLA 91), A. Paepcke Ed., ed New York: Association for Computing Machinery, 1991, pp. 197-211.
10. R. C. Martin, Agile Software Development: Principles, Patterns, and Practices. Upper Saddle River, NJ: Prentice-Hall, 2003.
11. T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, vol. SE2, pp. 308-320, 1976.
12. T. J. McCabe and C. W. Butler, "Design Complexity Measurement and Testing," Communications of the ACM, vol. 32, pp. 1415-1425, 1989.
13. P. M. Shanthi and K. K. Duraiswamy, "An empirical validation of software quality metric suites on open source software for fault-proneness prediction in object oriented systems," European Journal of Scientific Research, vol. 51, pp. 166-181, 2011.
14. V. Basili, L. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," IEEE Transactions on Software Engineering, vol. 22, pp. 267-271, 1996.
15. K. E. Emam, W. Melo, and J. Machado, "The prediction of faulty classes using object-oriented design metrics," Journal of Systems and Software, vol. 56, pp. 63-75, 2011.
16. W. Li and S. Henry, "Object oriented metrics that predict maintainability," Journal of Systems and Software, vol. 23, pp. 111-122, 1993.
17. J. M. Veiga and M. J. Frade, "Treecycle: a Sonar plugin for design quality assessment of Java programs (Technical Report: CROSS-10.07-1)," Fundação para a Ciência e a Tecnologia, Centro de Ciências e Tecnologias de Computação, Braga, Portugal2010.
18. G. Vandana, K. K. Aggarwal, and Y. Singh, "A fuzzy approach for integrated measure of object-oriented software testability," Journal of Computer Science, vol. 1, pp. 276-282, 2005.
19. ISO/IEC, Software engineering - Product quality - Part 2: External metrics (ISO/IEC TR 9126-2). Geneva: International Organization for Standardization and the International Electrotechnical Commission,, 2003.
20. T. M. Pigoski, Practical Software Maintenance: Best Practices for Managing Your Software Investment. New York: John Wiley & Sons, Inc. , 1997.
21. S. Khatri, R. S. Chhillar, and A. Sangwan, "Analysis of factors affecting testing in object oriented systems," International Journal on Computer Science & Engineering, vol. 3, pp. 1191-1196, 2011.

Paths of Adoption:

Routes to Continuous Process Improvement

David Saint-Amand, Naval Air Systems Command
Mark Stockmyer, Naval Air Warfare Center

Abstract. This paper covers the different types of teams the authors have encountered as NAVAIR Internal Process Coaches and how they approached Process Improvement with them, with special emphasis on the curmudgeons (bad-tempered, difficult, or cantankerous persons).

Introduction

There are many approaches to Continuous Process Improvement (CPI). The authors have observed, participated, or assisted in numerous CPI initiatives in both private industry and United States Federal Government service. Those efforts included Agile, Capability Maturity Model-Software (CMM®-SW), Capability Maturity Model Integration (CMMI®), Total Quality Management (TQM), High Performance Organization Training (HPO), Lean Six Sigma, Personal, Software Processes (PSP℠), Rational Unified Process (RUP), the Team Software Processes (TSP℠), and possibly a few others which may have been forgotten. All of these systems have a good chance for success if the people and organizations to which they are being applied are properly prepared, and the initiatives are managed in the same manner as a project. A good example of this would be the use of the ADKAR model, with its five objectives, to prepare for and execute a process improvement effort [1]:

- Awareness of the need for change
- Desire to support and participate in the change
- Knowledge of how to change
- Ability to implement desired skills and behaviors
- Reinforcement to sustain the change

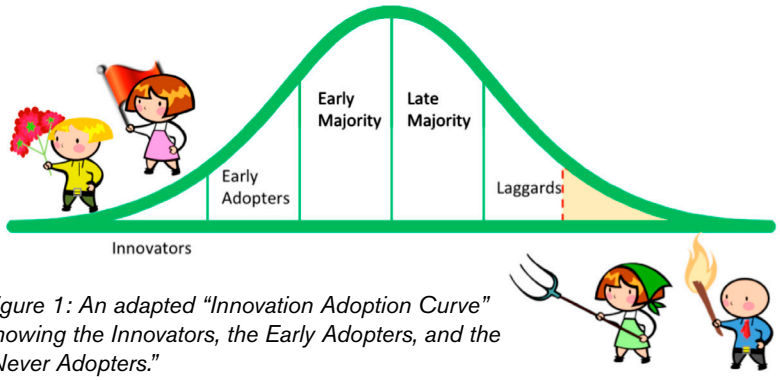


Figure 1: An adapted "Innovation Adoption Curve" showing the Innovators, the Early Adopters, and the "Never Adopters."

Another good example would be the use of the Software Engineering Institute (SEI) IDEAL model, IDEAL has five steps, the last four of which are repeated in a continuous cycle of process improvement [2]: Initiating, Diagnosing, Establishing, Acting, and Learning.

What can a process improvement coach do though when the proper preparations aren't made and the people are not effectively prepared in advance? If the success of a CPI initiative for any given team is defined as the whole team undertaking and sustaining CPI, then there are many paths to the adoption of CPI.

When preparing to introduce CPI in this sort of environment, it is important to remember that all pre-CPI teams are different. Some are made up of process champions, while others seem to have only arm-folded curmudgeons. Using our experiences as NAVAIR Internal process coaches we will address the following questions. How are these team types different? How can a process coach take these teams from ad-hoc processes to disciplined superstars? Are special approaches required? Most importantly, and this is where we spent most of our time as process coaches, what kind of techniques can you use to deal with the more difficult teams?

The authors believe that the answers they have found are applicable to most process improvement initiatives. While this article focuses on recent NAVAIR initiatives which utilized the Team Process Integration (TPI), it is because the TPI was able to provide them with data from which to draw conclusions about successful approaches to pursuing CPI.

The TPI is a NAVAIR derivative of the SEI's TSP. The TSP is a disciplined approach to writing software originally based upon the SEI Capability Maturity Model – Software (CMM-SW). The TPI takes the process scripts of the TSP and strips out the elements that are specific to software development. This leaves a generalized set of process scripts which may be customized and applied to support the planning, project execution, reporting, and process improvement efforts of both software and non-software teams.

To begin the discussion of the different types of teams let us briefly review the by-the-book approach to instituting the TSP familiar to its practitioners. We start with the "Innovators" and "Early Adopters" as defined in the "Innovation Adoption Curve" (Figure 1) [3]. They are willing to try new things, they work hard toward success, and they socialize that success to their friends. Their friends are encouraged to try the new techniques and the good new spreads from there. It has been the authors' personal experience in both the private software industry and the DoD that this is seldom the approach used. It has been more typical in these working environments for management to simply mandate all teams to use a new process improvement framework. While some teams may respond well to that approach, many do not. Whether they do or not often depends on the percentage of individual 'laggards' on a given team. If the team is made up almost exclusively of laggards, we call them "Never Adopters" and a special approach will be required.

Different Working Environments

The by-the-book approach, as described above, was developed by experts who hoped that organizations seeking process improvement would pursue it in an idealized, formal fashion. Some of the characteristics of that idealized approach are:

- the project is new
- the members of the project might never have worked

together before
- everyone, from management to the individual engineer, is properly trained and prepared
- the purpose of the weeklong Team Project Planning Session (aka "the Launch") is to build the team
- the organization has sufficient communication between projects to allow the "Adoption Curve Strategy" to work across the organization

This strategy has been used on numerous teams in the non-Academic environments of both the DoD and private industry. During the course of those efforts some differences between the Academic and non-Academic environments were identified. In general, in the non-Academic environments:
- the project is to enhance and maintain an existing product
- the team already exists and team members have often been working together for years
- in the interest of schedule, importance, and budget, not everyone is trained properly, if at all
- the Launch is used to introduce "Best Practices"
- few of the projects in a large organization talk to each other, making the dissemination of TSP/TPI a grindingly slow process

These differences in environment are important. Depending on the type of team, they can make a by-the-book approach unworkable.

Different Types of Teams

It is said that the first step to recovery is admitting you have a problem. Based on that, we have found that teams may be divided into two general categories, depending on who is doing the "admitting." Is it the team or their management?

Self-Selected Teams

These are the teams who know that something is wrong with the way the work is being done and they are personally motivated to do something about it. In our experience, these teams will usually identify one of three primary reasons to adopt process improvement:

1. To fix broken management: Irrational management has created a chaotic environment

2. To obtain a process the team will use: The Team Lead has worked on teams with good processes and wants their new team to start out on the right foot

3. To save the broken schedule: The product is chronically late and the team as a whole decides that they need a way to judge their progress.

Management-Selected Teams

If it is the management that decided there is a problem, which they think may be solved by process improvement; it is usually for one of the following reasons:

1. To fix the broken team: They have teams where the product is never delivered or, if it is, the product doesn't work

2. To introduce best practices: Management read about a process improvement framework in a White Paper

3. To gain insight into the schedule: Their teams are unpredictable and product delivery is never known until the last minute

It has been our experience that Management-Selected teams are typically neither convinced to, nor properly prepared to undertake process improvement. They are instructed to do it. Some teams will take this new effort on willingly, but the Never Adopters will not. The probability that these types of teams will be successful in the pursuit of mandated Continuous Process Improvement can vary widely.

Introduction Strategies for Self Selected Teams

Self-Selected teams, by their nature, are generally not difficult to deal with and can be expected to attempt to take on all the TSP or TPI practices from the start. A coach should not have to spend much time with them outside of what might normally be expected. These are the teams where the Innovation Adoption Curve works, and for which TSP coaches are trained.

In general, the team's chance of success is good, but not always.

Self-Selected: Fix the broken Management

Occasionally, an engineer from a disciplined team joins a new team and the new surroundings are not that for which they bargained. The engineer figures out too late that the new environment is not quite as disciplined as they had imagined, but at least they figure that they can control their own world.

There is a low chance of success here because it is likely that, intentionally or unintentionally, management itself is fostering an environment of chaos and will oppose attempts to introduce discipline.

Is there a good reason for a person to pursue process improvement in this environment even though the chance of success is low? Yes. That person builds a record of their attempt at discipline and that will serve them well later on when the project fails and management is attempting to assign blame. One of the authors of this article assisted an engineer in this situation, and because that engineer had planned, documented, and tracked their work, they were recognized by upper management as a competent, honest, and diligent employee: complaints lodged against him by his irrational manager notwithstanding.

Self-Selected: Obtain a Process the Team Will Use

In this instance the team lead is an innovator, an early-adopter type, or a person who may have formerly worked on a high-discipline team. They understand that chaos is a poor product development strategy. They want a team process and are willing to try new things. As the new team lead they have an opportunity during their new Team Lead "honeymoon" period to introduce CPI and they make the most of it.

The overall chance of success in adopting Process Improvement is high.

Self-Selected: Save the Broken Schedule

In this instance, an entire team comes to a consensus that something is wrong, be it schedule, cost, or quality. Not only do they admit they have a problem, they are willing to accept change in order to find a better way to do business.

The overall chance of success in adopting Process Improvement is high.

The Management-Selected Teams

Teams are usually selected for Process Improvement when management:

- Understands there is a problem with a team's performance
- Hears about other teams' success with process improvement
- Wants a process coach to fix their broken team

Management-Selected teams all respond to the new CPI initiative in pretty much the same way and are the process coach's biggest hurdle. They are often staffed with experienced professionals who have seen many Process Improvement flavor-of-the-month initiatives start and fail, or worse, start and be abandoned with the next change in management. If a process coach tries using the by-the-book approach with these teams, the probability of success is low. This is because the Canon of this approach often runs squarely into the Reality of the work environments of the DoD and Industry (Table 1). It is because these teams can be such a challenge for a process coach that the remainder of this article will focus on them.

It is likely for the Management-Selected teams that at least one member of the team with significant professional experience will be cynical and has learned that the best strategy for avoiding "work disruption" is to passive-aggressively resist the latest initiative. If all the members of the team fit that description then the process coach has entered the "Never-Adopters" portion of the Innovation Adoption curve. A coach will spend much more time with these teams than with a self-selected team, and the bulk of that time will be a seemingly endless effort to cajole the team into complying with the initiative.

For the Never Adopters, the introduction of process improvement must be done slowly and incrementally. Overwhelming them with process will just give them the ammunition they need in their complaints to management that what the process coach is asking cannot be accomplished and still expect them to get work done. Worse, the ferocity of their passive-aggressive resistance will blind them to any value there is from the process improvement initiative. At best, they will do the very minimum they can get away with and still be seen to be complying. In some cases they may even outright refuse to participate. Then, after the initiative collapses due to the team's intentional failure to perform, they will point to the lack of progress and data and say that the process is a hoax. They will say this even in the face of evidence of other team's successes with the same initiative. Typically their explanation is that their work is unique and not at all like the other team, whose success is either some sort of very-specific lucky break, or an outright lie.

In a further effort to justify their failure, they will spread this word throughout the larger organization and that will "poison the well." As a result, the organization will be less likely to try that brand of process improvement in the future and the coaching organization will lose other process improvement opportunities.

If management keeps up the pressure for process improvement, despite the manufactured failure and team complaints, then there is a risk of the organization losing valuable corporate knowledge through early retirements and lateral transfers.

So, if taking the by-the-book approach is likely to produce poor results, will rolling out process improvement in small doses really result in long-term success? If the goal is to change the culture and improve the practices of an organization, then the experiences of the NAVAIR Process Coaches suggest that the answer is 'Yes.'

Introduction Strategies for the "Never Adopters"

Everyone knows, from the manager who orders it, the process coach who has to introduce it, and the team who has to implement it, that they are eventually going to have to eat the entire process-improvement elephant. So, how do you get the "Never-Adopters" to undertake the effort? The key is to convince the team to agree to try a bite of the trunk, just to see what it tastes like.

Ask the team to:

- plan their work: introduce the team to projects with detailed plans
- track their work: start to instill process discipline
- think about Quality: get them to consider the possibility of building on the process

From this modest introduction, the process coach wants the team to come to understand that collecting performance data is neither difficult nor time consuming, that their performance data will not be used against them, and that there is value for them personally in the data which they are collecting.

Introduce Planning

Of all the process improvement practices this brings the greatest benefit, but it is not a common practice: have the team who will be creating the product build the project development plan. Many engineers in industry and the DoD have never seen a detailed plan, let alone participated in making one. The planning effort might not be considered much fun at first, but the resulting plan will be popular. Here are two quotes from the end of one such planning session:

"This is the first time I've known what I should be doing on this project."

Canon vs. Reality

Canon	Reality
Each project starts with a new team of people who have never before worked together.	The team is already established, sometimes for more than a decade.
The arduous effort of a detailed weeklong planning session will 'jell' a collection of strangers into a high-performance team.	The team is already jelled, and if they aren't, they soon will be as the process coach becomes their common enemy.
You roll out the entire process at once. The team is exposed to everything and is expected to put all of it into practice, improving steadily over time.	You can only get the 'difficult' teams to accept some small portion of the overall process.

Table 1: The by-the-book approach to Process Improvement and what a Process Coach is likely to encounter

"We should always have a plan."

Start the project team off with basic planning techniques. Have the team create a task list, estimate task size in terms of time and keep the workflows, if there are any, simple.

Introduce Tracking

Now the team has a plan, how are they going to track progress? Have them use a project tracking tool. There are a number of commercial programs available, and several of them are free to use. By using one of these tools, each member of the team will record their personal time against their tasks, and mark those tasks as completed when they are finished with them. Emphasize to the team that tracking their own work will, with very little effort on their part, provide them with personal insight into the following areas: Time on task, Earned value, Schedule progress, Forecasts, and Accuracy of estimates.

Introduce Quality

Will a team have good quality assurance practices right from the start? That is unlikely. Will they have a quality product? That depends. Their quality will probably be better than in the era before they started to make detailed plans, if only because their software interfaces are usually better defined. The key is that the process coach starts the discussion on quality which primes the team for introducing more disciplined quality practices later on.

Build on the Process

Will the team have high-quality personal data at the end of the first project cycle? Probably not. Most likely they weren't too diligent in recording their own data, but it is still data. After the first cycle, the team members will begin to see that their plans have useful information in them, and they will see that the data wasn't as good as it could have been. Most engineers like data and the desire for better data encourages them to improve the way in which they have been tracking their effort. It also leads them to begin wondering "if this data is useful, what else might I track that would be of interest?" They begin to think "if some process isn't bad, more process might be better." It is in this way that individual members take themselves from 'Process Resistors' to 'Process Defenders' and then on to 'Process Advocates.' Once that starts, the team is on the road to higher performance.

Team Results Over Time

So, the strategy outlined for introducing CPI to Never-Adopters is to start them out with simple processes and build on them over time. It will certainly take longer, but will it actually produce positive results? The answer is yes, as the results of the efforts of two different types of TPI teams will show: a team of software testers, and an Interdisciplinary team of Software, Electronic, and Mechanical engineers.

Team A: The Software Test Team

Figure 2 shows how a team of software product testers fared over time in their tracking of the actual hours associated with their Task Time. The first chart shows the data for the first TPI cycle, and the second for the fourth TPI cycle: a span of two years. The red lines represent the planned accumulation of task hours as estimated during the launches. The blue lines are the actual number of task hours as logged by the team during the project cycles.

If you move the Actual line of the fourth project cycle to the left to account for the delay in the start of testing, you can see that the team is accurately estimating their availability to work on the effort.

Figure 3 shows how the team fared in tracking their earned value (EV) over the span of the same four project cycles. The red lines represent what was planned at the launches. The blue lines are the EV that the team accrued during the project cycles. The green lines are the actual cost of that EV in terms of hours.

While the actual EV never matches the EV progress as anticipated by the planning tool, the actual EV and the actual cost of that EV are very close. As a result of using the TPI, this team is able to accurately estimate the size of their tasks, even though they were estimating solely on the basis of time.

Team B: The Interdisciplinary Team

How well did this approach work for the interdisciplinary TPI Team composed of software, electronic, and mechanical engineers? The results can sometimes be startling. Figure 4 shows the team's performance during their first TPI cycle. The red line represents the planned accumulation of task hours as estimated during the launch. The blue line is the number of task-hours as logged by the team.

It is all the more impressive as they had only one day of TPI training.

The outcome is equally exciting for their EV tracking (Figure 5), where the tasks were, for the most part, simple tasks estimated in units of hours or days, not Source Lines of Code (SLOC) or some other more direct measurement. The red line represents what was planned at the launch. The blue line is the EV that the team accrued over time. The green line is the actual cost of that EV in hours.

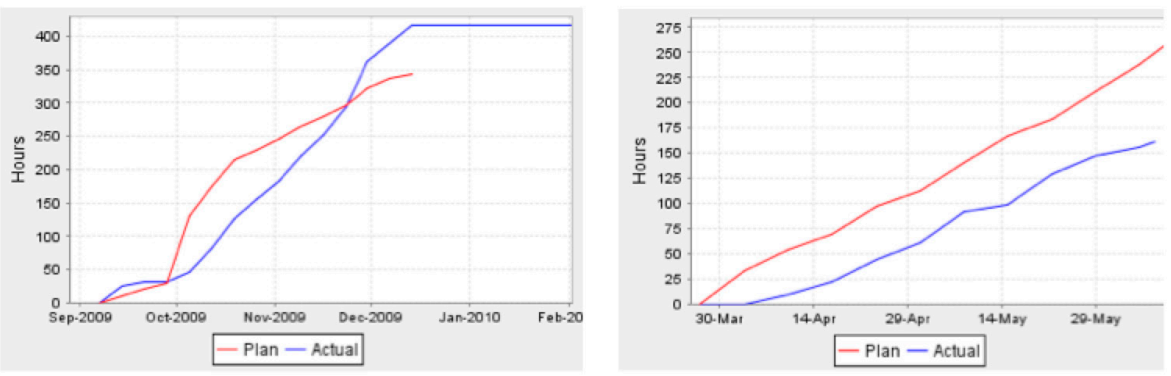


Figure 2: Direct Time charts for a team of software testers: the first cycle on the left and the fourth on the right.

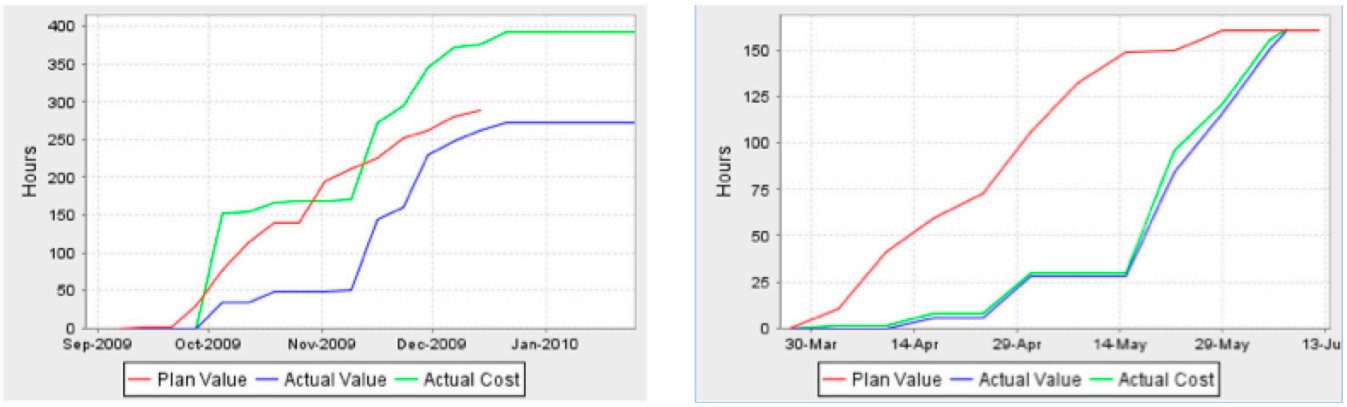


Figure 3: Earned Value charts for a team of software testers: the first cycle on the left and the fourth on the right.

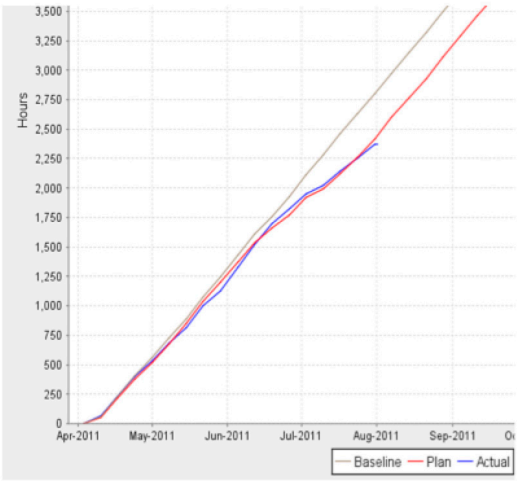


Figure 4: Direct Time chart for an Interdisciplinary team of engineers from their first project cycle.

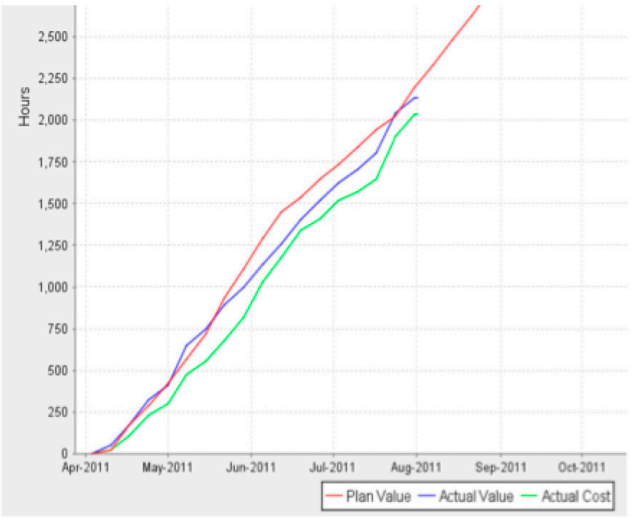


Figure 5: Earned Value chart for an Interdisciplinary team of engineers from their first project cycle.

First Cycle Comments	Second and Third Cycle Comments	Fourth Cycle Comments
Launch was dreaded by everyone	More comfortable with the process and the plan let me know what to work on next	Liked having historical data. Made the Post Mortem less painful than in the past
"What have we done to ourselves?!"	Liked consulting, designing, and planning together	Work patterns are emerging
The Launch is one more thing taking time out of my availability for work	Injected discipline into work. Helped to keep focus	Need more rigorous planning requirements
Not nearly as bad an experience as I thought it would be. Turned out to be relatively painless	Interesting to see the kind of statistics being collected	Stopped launch tasks to work out issues and sync team understanding
The Project Launch was efficient and effective	Emphasized the importance of logging time as you go, instead of back filling	The team lead and planning manager are spending less time on preparing the monthly management report as the necessary information is readily available
The Coach accepted that it was more important to start measuring the existing process rather than force the team to adopt practices that the team will probably not do	Kept the Coach employed	
Next time we should have more detail on the number and type of Development Tasks		

Table 2: Typical comments from the first project cycle. Color added for emphasis.

What About Quality?

It has it been our experience that convincing the never-adopter teams to perform quality assurance practices, other than the usual test-and-fix, has been one of the most difficult areas of our CPI efforts. At the time that this article was written, some of our teams had on their own initiative taken on peer reviews, and they do seem to be more open to additional quality control practices, but they are just getting started on this part of their journey. It is too early yet to know what sort of progress to expect.

Changes in Attitude

It was hinted at earlier in this article that if a process coach took the incremental CPI path, the team members' attitudes towards process improvement would become more positive over time. The evidence for that change in attitude may be found in a comparison of selected comments collected from the launches and postmortems of four six-month project cycles of four TPI teams (Table 2).

The teams went into their first project cycle launch with the idea that it would be a useless, miserable experience. They left feeling that:

- it wasn't unbearable
- they had some control over their work
- the plan generated during the launch was their plan
- they would like to have had more detail in the plan

By the Second and Third project cycles the launches are taking less time, and now that they know what to expect, are beginning to seem easy. More importantly, to the process coach anyway; the coach has gone from being seen as the common enemy to being part of the work environment. In essence they:

- worked together as a team and enjoyed it
- found the rigor of the new processes to be beneficial
- liked that there was data to analyze
- wanted better data from the next cycle

The Fourth cycle comments suggest that after two years of following the incremental CPI path, the project launches are now easy, safe, and relatively fun. The teams:

- have historical data they can use to estimate their future work
- are beginning to take control of their current work
- are working together to create the plan and iron out the unclear parts
- understand that process improvement is saving them time-and-effort

These results are evidence of a strong improvement in team's attitudes towards CPI.

Final Comments

The long-term goals of Process Improvement should be to introduce and sustain a culture of continuous process improvement. The results of the incremental approach used by the authors suggest that not all teams have to take the steep path towards that goal. After several years of coaching Never-Adopter teams, NAVAIR Process Coaches have seen steady improvement in the ability of their TPI teams to estimate their level of effort and schedule, and have seen positive changes in team member's attitudes towards process improvement. While the journey for these teams is not yet over, it appears that by taking the slow, incremental path, reluctant teams may be able to make themselves into process-improvement-oriented teams which actively search for ways to do business better.

Disclaimer:

CMMI,® CMM,® PSP,℠ and TSP℠ are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

ABOUT THE AUTHORS

David Saint-Amand is a Team Process Integration (TPI) Coach with the Naval Air Systems Command Process Resource Team. His previous positions include DCS Corporation Section Manager, Naval Operations Research Analyst, and Engineering Geologist and Seismic Safety Consultant. He holds a B.A. in Geology from the University of California at Santa Barbara with a secondary emphasis in Computer Science. He is a Defense Acquisition University Certified Level III Life Cycle Logistician and an SEI-Authorized PSP Instructor.

Mail Stop 6308, 1900 N. Knox Rd.
China Lake, California 93555-6106

Phone: 760-939-2372
FAX: 760-939-0150
E-mail: David.Saint-Amand@navy.mil

Mark Stockmyer is the Software Lead for the OASuW (Offensive Anti-Surface Warfare) Technical Project Office at the Naval Air Warfare Center in China Lake, California. His previous positions include Fire Control System software engineer for the SPIKE missile project and systems software engineer at TouchNet Information Systems, Inc. He holds an M.S. degree from the University of Kansas in Computer Science and B.S. degrees from Missouri Western State University in Computer Science and Chemistry. He is an SEI-Authorized PSP Instructor.

Mail Stop 6612, 1900 N. Knox Rd.
China Lake, California 93555-6106

Phone: 760-939-3979
E-mail: mark.stockmyer@navy.mil

REFERENCES

1. Hiatt, Jeffrey M. ADKAR: A model for Change in Business, Government and our Community. Loveland, CO: Prosci Research, 2006.
2. McFeeley, Robert; IDEAL: A User's Guide for Software Process Improvement. Software Engineering Institute, Carnegie Mellon University, 1996.
3. Rogers, E. M. Diffusion of innovations (3rd edition). New York: Free Press, 1983.

High Maturity Heresy: Doing Level 5 Before Level 4 Without Data

Tom Lienhard, Raytheon Missile Systems

Abstract. Where does our knowledge regarding High Maturity of the CMMI® come from? Usually from CMMI training classes, CMMI conferences, CMMI lead appraisers, consultants and other CMMI "experts." And how can we forget the "upfront material" of the CMMI and the infamous page 80 of CMMI Ver 1.1? What if all these sources were wrong or, at best, were only painting half the story? After all, these sources all stem from the same origin.

My personal evolution of High maturity understanding is depicted in Figure 1 below. It started with the Software CMM® (SW CMM) where high maturity was usually tied to the statistical control of defects found in peer reviews. I later became a BlackBelt and learned all about variation and the analysis of variation. From there I was exposed to the CMMI and attended the Understanding High Maturity Practices class at SEI, where I learned that High Maturity was about control charts. It was not until I truly understood that taking advantage of High Maturity Practices is about identifying business objectives, and what influences achieving those objectives, that I was able to pull my knowledge together and fully comprehend the potential of an organization that implements High Maturity Practices.

Raytheon Missile Systems (RMS) achieved SW CMM Level 5 in 2001 by statistically controlling defects detected in the software development process. Improvement was realized and return on investment was made however programs were stilling having the problem of being able to produce product at a price the customer was willing to pay. So, what went wrong, RMS was Level 5?

Was the objective of High Maturity to indentify an iterative process so statistical process control (SPC) could be applied? Was it to hang a Maturity Level 5 sticker on the wall? Or was it about identifying true business objectives and the key processes that impact those objectives, then statistically controlling those key processes to maximize the probability of meeting the objectives? We needed to step back, understand the key business objectives and concentrate on achieving those objectives. Not simply following what had been up until now, "success" in achieving High Maturity.

RMS could be described as a "high-volume prototype" factory. Although RMS builds families of weapons (missiles, projectiles, etc) each has their own unique objectives. Some are surface launched, some are launched from aircraft. Some have rocket motors, some glide. Some are small, some are large. Some are guided by GPS, some by laser and some by an Inertial Measurement Unit (IMU).

RMS needed to change their approach from one of designing a product, implementing that design, and then re-designing the product because the design was too expensive or could not be built in the volume needed to an approach of understanding the intended use of the product, making capability trades around affordability and produceabilty, and then designing the product to maximize affordability and produceabilty, as shown in Figure 2.

The paradigm shift was necessary because a business cannot survive if they design technically excellent products that can't be produced at a price the customer is willing to pay. Analysis

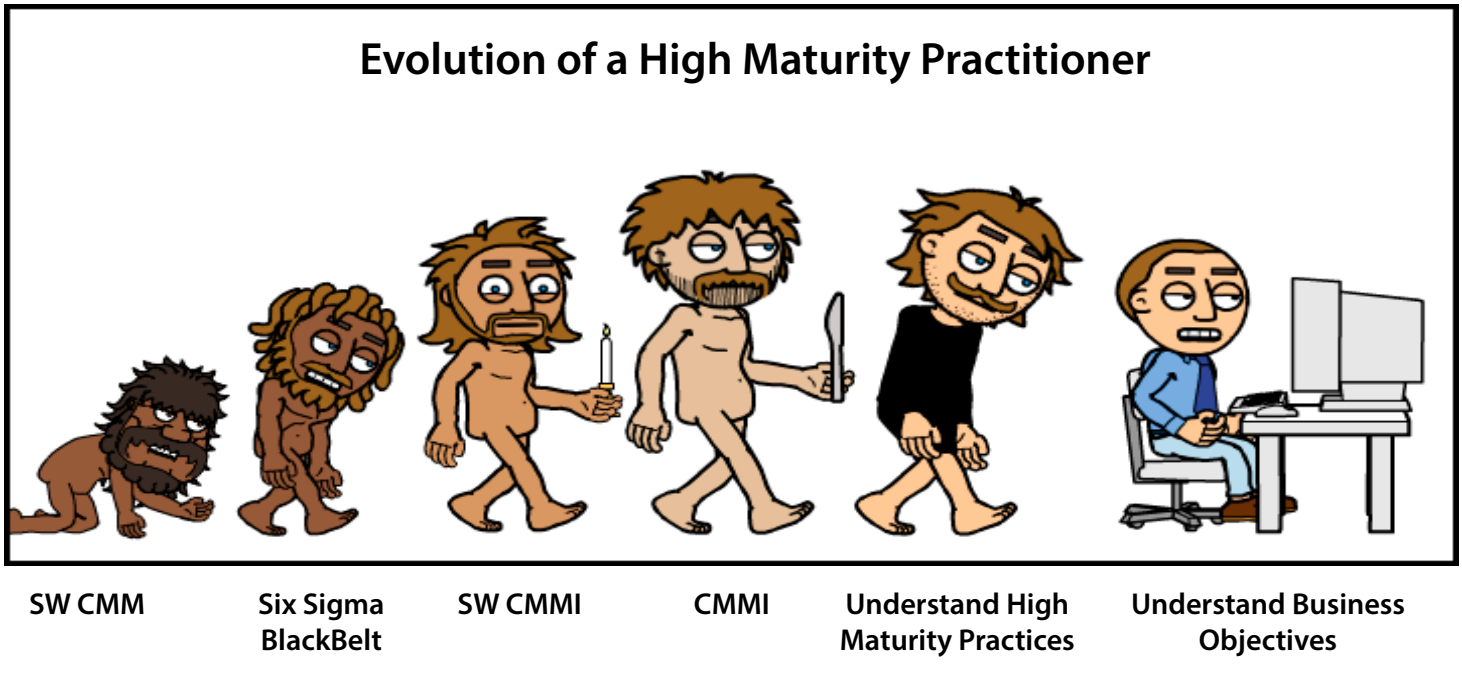
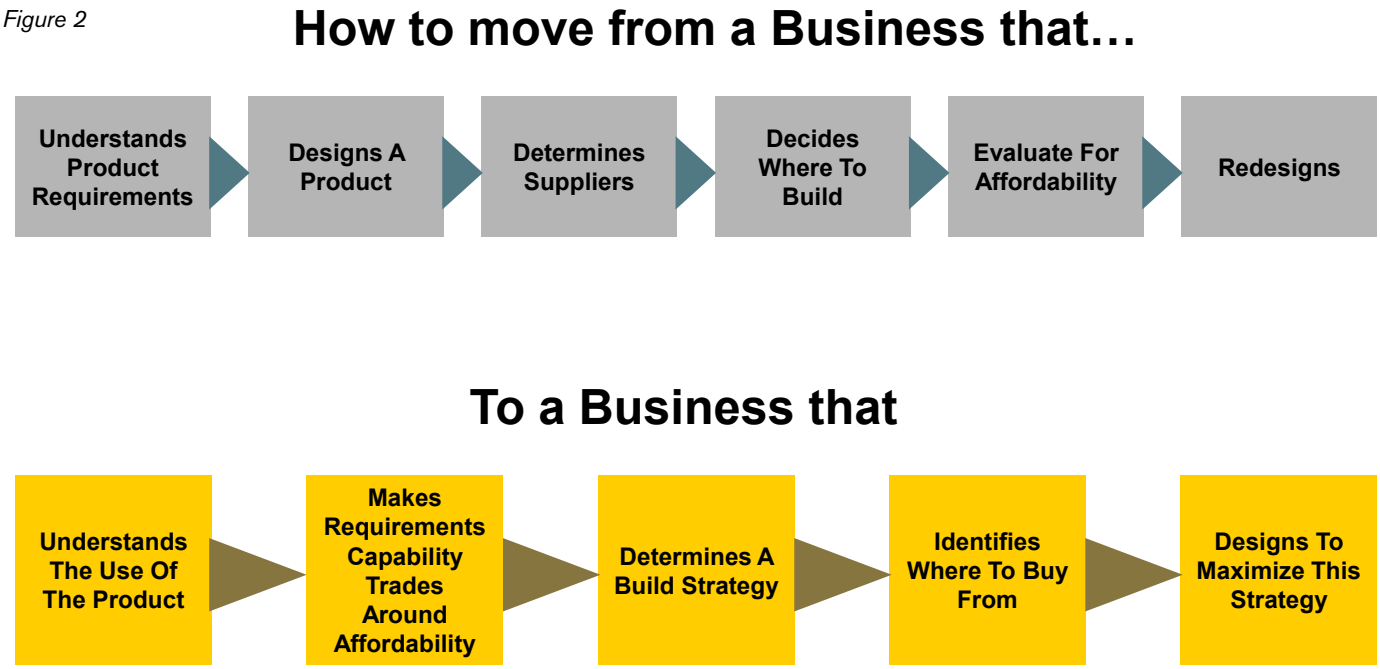


Figure 1: My Personal Evolution of High Maturity Understanding

Figure 2



showed that 70% of product lifecycle costs were determined prior to the start of development yet over 75% of the cost is spent post development. In other words, once the design is on paper, over 70% of the cost is locked in.

The ah-ha moment came when RMS realized that the product lifecycle was expanded when the product was more than software, see Figure 3. SW CMM caused the lifecycle to be seen as development, since there was no manufacturing associated with software. When the SW CMM was sun-setted and the CMMI took over, it was easy to duplicate the software solution for high maturity (statistically control the peer review process), replicate for systems and hardware and claim victory. But keeping the status quo, focusing on just the development lifecycle, would completely miss RMS' business objective – to reduce the Average Unit Production Cost (AUPC), reduce scrap, and increase yield.

Focus needs to be on the entire product lifecycle, from pre-concept through production, not just development. If the focus is just on optimizing the development lifecycle, it might actually increase the overall lifecycle costs. Or worse, negatively impact the business objectives, e.g. increase AUPC, increase scrap, and decrease yield. Production is ultimately where RMS will make a profit or lose their shorts. A small savings per unit in production can add up to be far greater than the entire development cost, refer to Figure 4. RMS was caught in the paradigm that the SW CMM, CMMI-Dev and industry caused – focusing High maturity Practices on development.

This was an epiphany. No longer think of the CMMI in terms of software, hardware, and systems but in terms of System Development. Remember what is critical to the RMS' business. Production needs to be the emphasis over development. Production is where cost and time is either minimized or super-inflated. RMS is willing to invest more resources in development in order to streamline production. When dealing with software, production is virtually "CTRL-C" and rarely impacts design decisions. Production is extremely complex with hardware and is very much impacted by design decisions. The scope of the lifecycle does not stop at the end of development but should include manufacturing (production) and fielding. There was a profound shift in focus from the typical Software Development 1st, Systems and Hardware Development 2nd to Production 1st and Development 2nd.

In the 1950s SPC was applied to product. Starting in the 1980s, in part thanks to CMM and CMMI, SPC started to be applied to processes during development. What RMS is doing in the 2010s is looking at the mission objectives of the fielded product in the pre-concept phase and developing process and product performance models to predict the capability of the process to produce products that meet the customers' needs at a price they can afford. This enables RMS to compose a defined process that

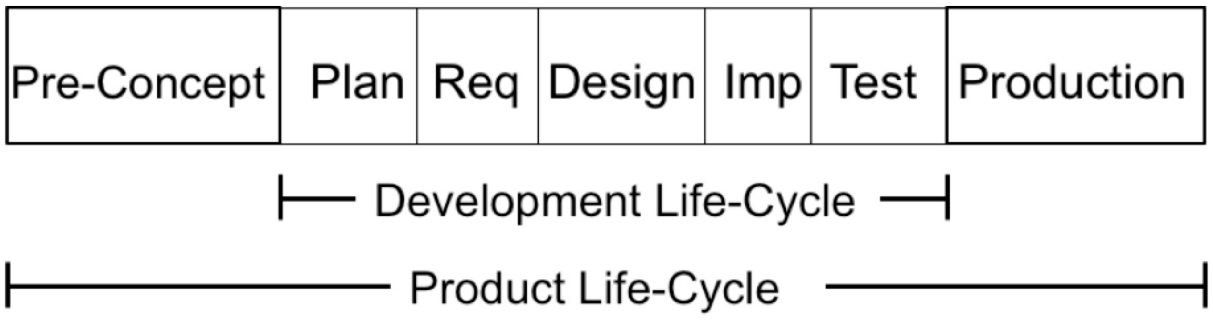
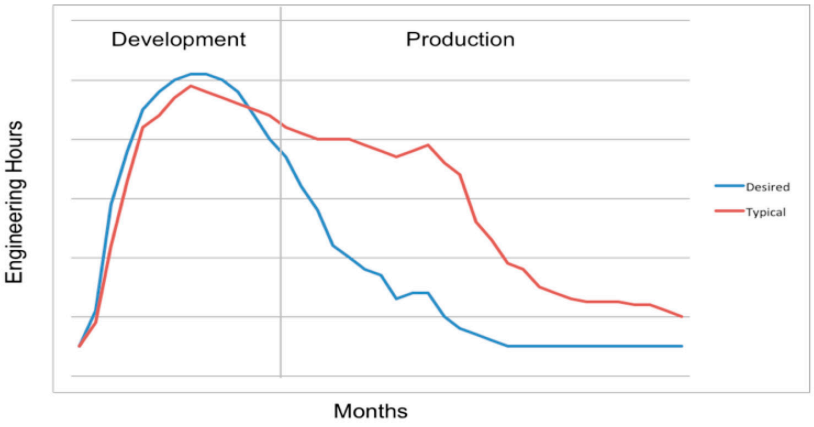


Figure 3

Figure 4



maximizes the probability of meeting the key objectives and a design that will not need to be redesigned once it transitions into production, see Figure 5.

The process and product performance models allow predictions to be made throughout the entire lifecycle by different groups for different purposes as shown in Figure 6. Beginning in the pre-concept phase, models and historical baselines from systems which have previously been built are used to determine if the concept is even feasible. These models gain fidelity as they progress through the lifecycle. When actual data becomes available, the models are recalibrated. In development, these models are used to predict performance, producibility and affordability, and optimize the design prior to "bending metal." During production RMS transitions from using models and simulations to SPC. The collections of models and resulting baselines are captured across the business for future programs to leverage and the cycle begins again.

For RMS, the goal is to balance performance, producibility and affordability to design a product which meets the customers' needs at a price the customer can afford. This is embedded in RMS' common process and is institutionalized via a plethora of statistical tools and techniques contained in the Raytheon Six Sigma Toolbox, including Quality Functional Deployment, Sensitivity Analysis, Design of Experiments, Reliability Predictions, Design for Manufacturing Analysis, Process Modeling, Producibility Assessments, and Cost as an Independent Variable and Process Capability Analysis.

Once the true business objectives are understood, the first step is to identify and understand what the customer needs. After that is established, transfer functions (or models) can be developed using tools including Process Capability Analysis Toolset (PCAT), Design Capability Analysis Tool (DCAT),Design and Analysis of Simulation Experiments (DASE) and Raytheon Analysis of Variation Engine (RAVE). These models can be used to help identify the influential factors or key characteristics that have the greatest impact on the customer needs. (In the CMMI world, these will be the subprocess that will be statistically controlled). These models can then be used to perform "what-ifs" and the knobs can be turned to determine where to set these key characteristics to maximize the probability of meeting the customer needs. These setting are captured and a control plan is established and used during production to ensure the key characteristics maintain within range. This is iterated at each subassembly and component level as appropriate.

This process helps eliminate over-design (high cost) and under-design (high scrap, rework and low quality) to find the sweet spot allowing RMS to design a product which meets the customers' needs that can be affordably produced. This is done starting at pre-concept through production. In CMMI terms, the

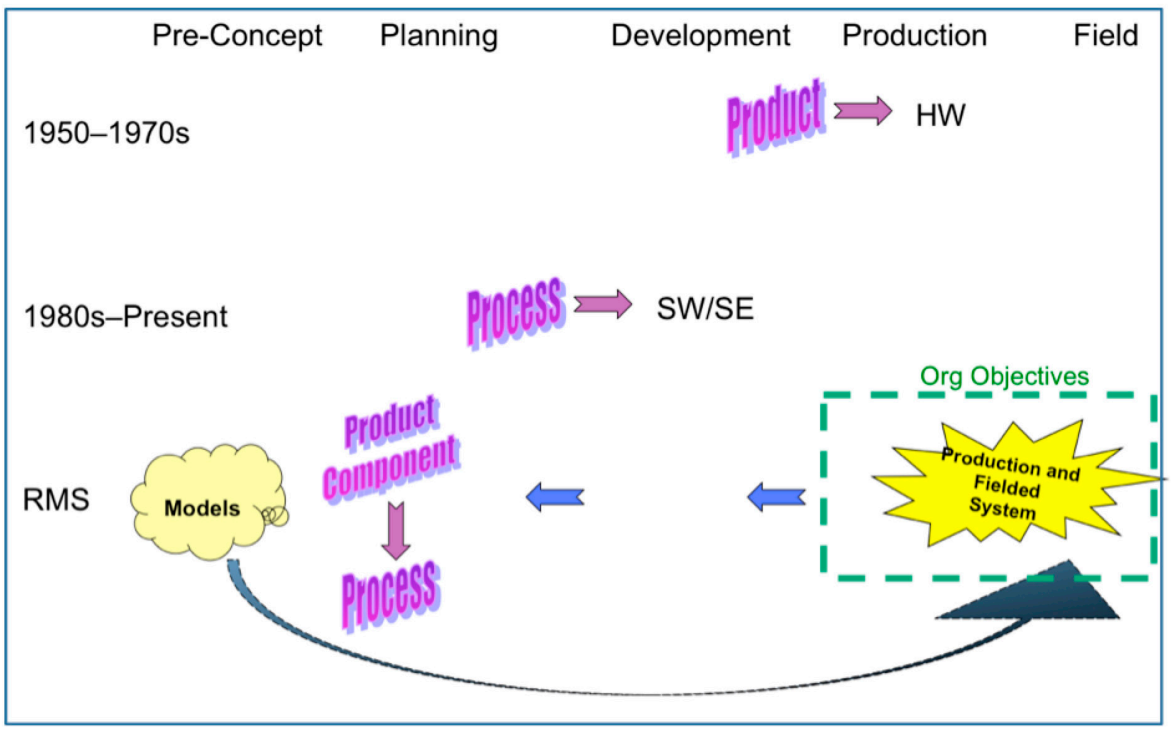


Figure 5

programs are predicting performance and optimizing the design using models and simulations prior to design and development without collecting actual measures, aka Doing Level 5 Before 4 without data!

Disclaimer:

CMMI® and CMM® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

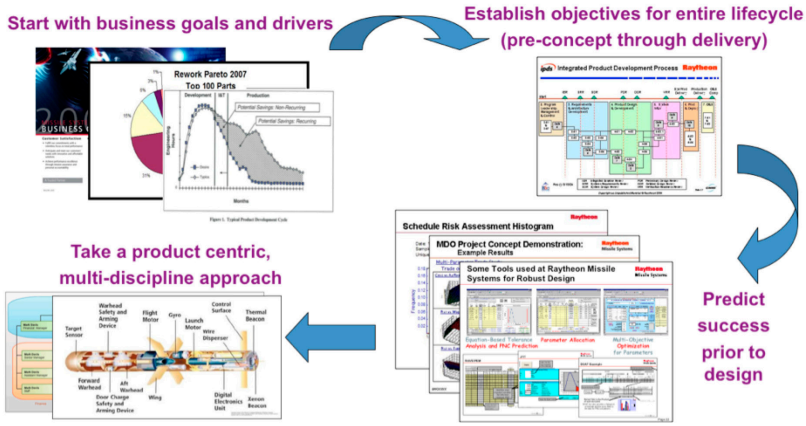
ABOUT THE AUTHOR

Tom Lienhard is a Sr. Principal Engineer at Raytheon Missile Systems and a Six Sigma Black Belt. Tom has participated in more than 50 CMMI® and CMMI® appraisals both in DoD and Commercial environments across North America and Europe and was a member of Raytheon's CMMI® Expert Team. He has taught Six Sigma across the globe, and helped various organizations climb the CMM and CMMI maturity levels, including Raytheon Missile Systems' achievement of CMMI® Maturity Level 5.

He has received the AlliedSignal Quest for Excellence Award, the Raytheon Technology Award and the Raytheon Excellence in Operations and Quality Award. Tom has a BS in computer science and has worked for Hughes Aircraft Co., Raytheon, AlliedSignal, Honeywell and as a consultant for Managed Process Gains.

Phone: 520-663-6580
E-mail: thomas_g_lienhard@raytheon.com

Figure 7



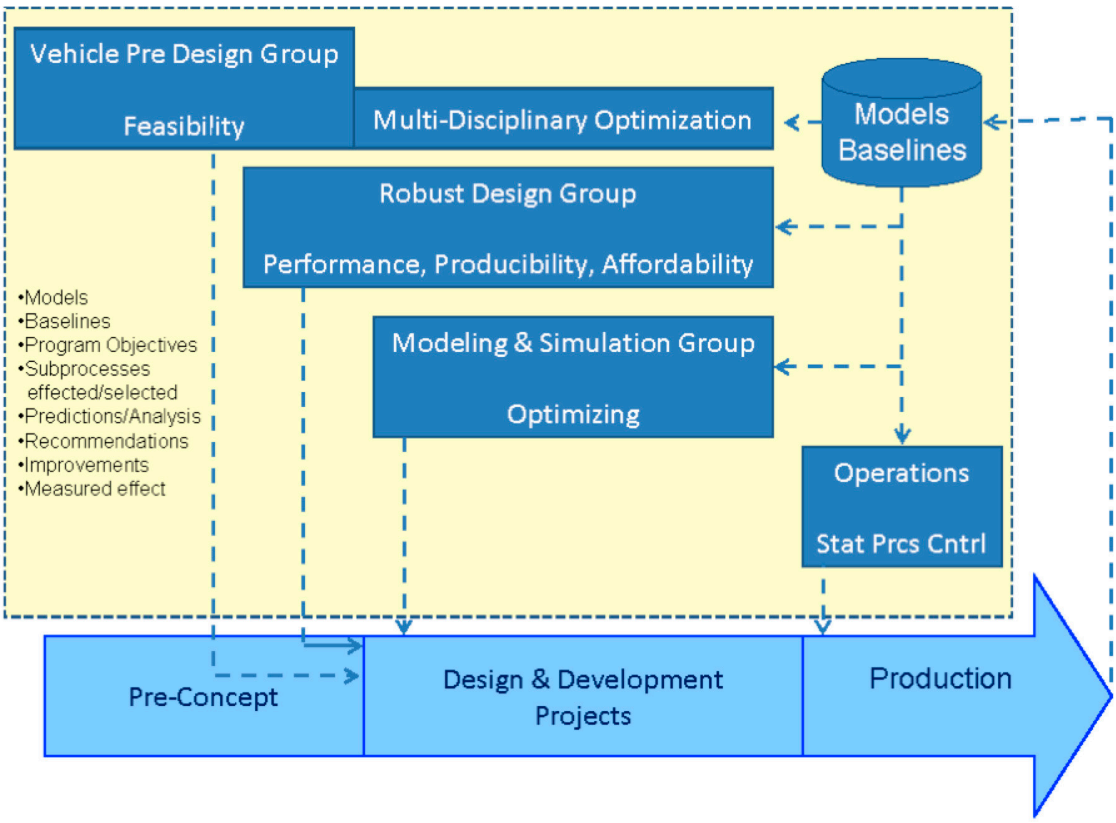


Figure 6

Upcoming Events

Visit ‹http://www.crosstalkonline.org/events› for an up-to-date list of events.

The 26th International Conference on Software Engineering and Knowledge Engineering
1-3 July 2014
Hyatt Regency, Vancouver, Canada
http://www.ksi.edu/seke/seke14.html

SPIN 2014: International SPIN Symposium on Model Checking of Software
San Jose, California, USA
21-23 July 2014
http://spin2014.org

Association of Software Testing: The Art of Science and Testing
August 11-13, 2014 New York City
http://www.associationforsoftwaretesting.org/conference/cast-2014/

SIGCOMM'14 – ACM SIGCOMM 2014 Conference
17-22 Aug 2014
Chicago, Illinois
http://www.sigcomm.org/events/sigcomm-conference

SEFM- Software Engineering and Formal Methods
1-5 Sept 2014
Grenoble, France
http://sefm2014.inria.fr/

APPSEC USA 2014
16-19 Sept 2014
Denver, Colorado
http://2014.appsecusa.org/2014/

International Conference on Software Engineering and Technology
17-18 September 2014
Paris, France
http://www.icste.org/

QSIC 2014 Int. Conf. on Quality Software
Dallas, Tx
October 2-3, 2014
http://paris.utdallas.edu/qsic14

SEDE 2014: The 23rd International Conference on Software Engineering and Data Engineering
New Orleans, Louisiana
13-15 Oct 2014
http://www.wikicfp.com/cfp/servlet/event.showcfp?eventid=33994&copyownerid=9837

17th Annual Systems Engineering Conference
27-30 Oct 2014
Waterford Springfield
http://www.ndia.org/meetings/5870/Pages/default.aspx

SUBSCRIBE TODAY!

To subscribe to CrossTalk, visit www.crosstalkonline.org and click on the subscribe button.

Schadenfreude and Mature Software Development

Ever heard of the word "schadenfreude?" It is what is called a "loanword" entering the English language from German. It can best be described as that delicious feeling you get when you see somebody cut you off at a four-way stop, but then immediately run into something. Technically, it is defined as pleasure derived from the misfortunes of others?" It is a very interesting word—it has a diverse background (being found in many languages), and various studies have shown the feeling of schadenfreude is linked to envy, possibly linked to a particular sex (some studies show that men feel it more), and also possibly linked to a feeling of low self-esteem. However, I personally think that that many/most/all software practitioners get some delight as seeing (or reading) about a colossal software failure.

Way back in 1996, a very good friend of mine, then-Captain Thomas Schorsch (now Retired Lt. Col, Ph.D.) was a student at the Air Force Institute of Technology. He wrote an article called, "The Capability Immaturity Model," published in CrossTalk (November 1996, a copy available at http://cs.hbg.psu.edu/comp413/cimm.pdf). In it, he described additional negative CMM® levels describing software immaturity. I wish I could turn back time—so I could convince Tom to let me be his co-author. I am in awe of the cynicism, sarcasm, and validity of Tom's research.

But such cynicism and sarcasm did not, unfortunately, start with Dr. Schorsch. Back in the early 1970s I had the following posted on my wall when I was an applications programmer back at Strategic Air Command:

Six Phases of A Software Project

1. Enthusiasm
2. Disillusionment
3. Panic and hysteria
4. Search for the guilty
5. Punishment of the innocent
6. Praise and honor for the nonparticipants

This list was probably not new even back in the 1970s. I can find similar sayings on the web, and this particular list was located at http://en.wikipedia.org/wiki/Six_phases_of_a_big_project). I am certainly not the originator.

In 1997, Robert Glass published a book entitled "Software Runaways: Monumental Software Disasters." I have a copy, and enjoy reading it over from time to time. You can almost feel the joy when you read about such massive failures. In fact, to quote from the Amazon.com "blurb" on the book, Runaways brings a software engineer's perspective to projects like: American Airlines' failed reservation system, the 4GL disaster at the New Jersey Department of Motor Vehicles, the NCR inventory system that nearly destroyed its customers, and the next-generation FAA Air Traffic Control System that collapsed.

I really enjoyed reading the book the first time I read it, and as I said, I try to re-read it yearly (enjoying it just as much). In fact, I am pretty sure I know why I enjoy reading about spectacular failures – there are two reasons: I was not part of the project, and, there are valuable lessons to be learned from spectacular failures.

The very next year, Glass followed up with a similarly great book, "Computing Calamities: Lessons Learned from Products, Projects, and Companies That Failed." I enjoy re-reading this book, also! You see, Robert Glass understands – sometimes the only benefit from failure is that you can learn from it. Sometimes being a bad example is the last, great act of a failing software project. There is no real joy or pleasure in seeing a list of failures and associated costs. There is, however, a lot to be learned from seeing how the failure developed, what steps were ineffectively implemented to stop the failure, and what the final straw was that broke the software engineer's back!

And that's the thing: I am not REALLY enjoying the massive failures of others. What I am appreciating is that I can learn from their failures without having to actually undergo the failure myself. I don't want to be part of a multi-million dollar failure. However, I certainly appreciate "lessons learned" that allow me to effectively reason "Wow – what's happening to me is what happened in the massive XYZ failure, and they tried this to fix it, and it didn't work. Maybe I better try something else."

I recently saw—for the umpteenth time—the movie Apollo 13. I love the part where Ed Harris, playing Gene Kranz (Flight Director) says, "Failure is not an option!" In spite of massive failures, they managed to bring the Apollo 13 crew home safe, using slide rules for complex calculations. Unfortunately, that was hardware, and this is software. Failure for software projects is, unfortunately, almost always an option—from the simplest printer driver to complex flight software. To make software work, it takes a lot of hard work, process discipline, and adherence to standards, directives, and regulations. It takes a lot of research on "lessons learned" from other projects. It also takes best practices, good lifecycle selection, and great designers making workable designs—architectural, data, interface and module design.

And—maybe learning a bit from other, similar projects that failed.

All of this takes high maturity. As I frequently tell my students, anybody can write code. Want to craft software instead? That takes maturity and discipline.

Disclaimer:

CMM® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

David A. Cook, Ph.D.
Stephen F. Austin State University
cookda@sfasu.edu